Lauri Vahlman

3D-grafiikkamoottori mobiililaitteille

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

11.5.2014

Metropolia

Tekijä Otsikko Sivumäärä Aika	Lauri Vahlman 3D-grafiikkamoottori mobiililaitteille 33 sivua + 4 liitettä 11.5.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro Tuntiopettaja Heini Puuska

Tässä insinöörityössä käydään läpi mobiililaitteille suunnatun yksinkertaisen 3D-grafiikkamoottorin suunnittelu ja toteutus käyttäen OpenGL ES -rajapintaa. Työssä esitellään grafiikkamoottorin toteutuksessa käytettyjä tekniikoita sekä tutustutaan moottorin rakenteeseen ja toteutuksellisiin yksityiskohtiin. Työn alkupuolella tutustutaan myös modernin 3D-grafiikan yleisiin periaatteisiin ja toimintaan sekä käydään läpi 3D-grafiikkaan liittyviä suorituskykyongelmia.

Työn loppupuolella esitellään yksinkertainen tekniikkademo, joka koodiesimerkein näyttää, miten grafiikkamoottoria voidaan käyttää käytännössä. Tämän lisäksi grafiikkamoottorilla suoritetaan suorituskykymittauksia, joiden perusteella arvioidaan moottorin rakenteen onnistumista sekä moottorin skaalautuvuutta.

Suorituskykymittausten perusteella grafiikkamoottorin todettiin olevan varsin onnistunut. Moottorin rakenteesta tuli selkeä, mikä tarjoaa hyvän pohjan moottorin jatkokehitykselle.

Avainsanat	Grafiikkamoottori, 3D-grafiikka, OpenGL

Author Title Number of Pages Date	Lauri Vahlman 3D Graphics Engine for Mobile Platforms 33 pages + 4 appendices 11 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Heini Puuska, Lecturer

This thesis describes the design and implementation process of a simple 3D graphics engine for mobile platforms using OpenGL ES. The technologies and techniques used in developing the engine are explored, and details of implementation and the structure of the graphics engine are revealed. At the beginning there is also general information about the workings of modern 3D graphics and about the possible performance problems that are often encountered when working with real-time 3D graphics.

At the end a small tech-demo that is used for showing how the graphics engine can be used in practice is presented. This is done through the use of code examples. The performance of the graphics engine was measured with a few simple test cases and the results were used for determining how successful the design of the graphics engine really was.

From the results of the performance measurements it was determined that the performance of the graphics engine was adequate. The overall structure of the engine was also determined to be rather successfull and offers a fine platform for further development.

Keywords	Graphics engine, 3D graphics, OpenGL

Sisällys

1 Johdanto

Mobiililaitteiden grafiikkaprosessorien suorituskyky on kasvanut valtavasti viimeisen parin vuoden aikana. Tämä suorituskyvyn kasvu on mahdollistanut yhä monimutkaisemman ja yksityiskohtaisemman reaaliaikaisen 3D-grafiikan käyttämisen mobiiliapplikaatioissa. Monet suosituimmista mobiilipeleistä ovat täysin kolmiulotteisia, joiden graafinen näyttävyys lähentelee jo edellisen sukupolven konsolipelejä. Myös monet mobiililaitteiden hyötysovellukset hyödyntävät 3D-grafiikkaa, esimerkiksi karttasovellukset ja erilaiset lisättyä todellisuutta käyttävät sovellukset (eng. *augmented reality*).

3D-grafiikan hyödyntäminen omassa sovelluksessa vaatii joko matalan tason grafiikkarajapintojen käyttämistä, tai sitten voidaan käyttää korkeamman tason grafiikkakirjastoja tai valmiita pelimoottoreita.

Tässä insinöörityössä toteutetaan yksinkertainen 3D-grafiikkamoottori käyttäen OpenGL ES 2.0 -rajapintaa. Moottoria testataan Android-laitteella, mutta periaatteessa sitä voidaan käyttää millä tahansa OpenGL:ää tukevalla alustalla hyvin pienin muutoksin. Moottorin suunnittelussa ja toteutuksessa pyritään ennen kaikkea selkeyteen suorituskykyä unohtamatta. Tämän takia moottorin tukemien ominaisuuksien määrä rajoitetaan pieneksi.

Työn alkuosassa tutustutaan 3D-grafiikan yleisiin periaatteisiin, tarkastellaan OpenGL-rajapintaa sekä tutkitaan 3D-grafiikassa usein esiin tulevia pullonkauloja. Sen jälkeen esitellään grafiikkamoottorin suunnitelma ja toteutukseen liittyviä yksityiskohtia. Lopuksi grafiikkamoottorin suorituskykyä mitataan ja arvioidaan sen skaalautuvuutta.

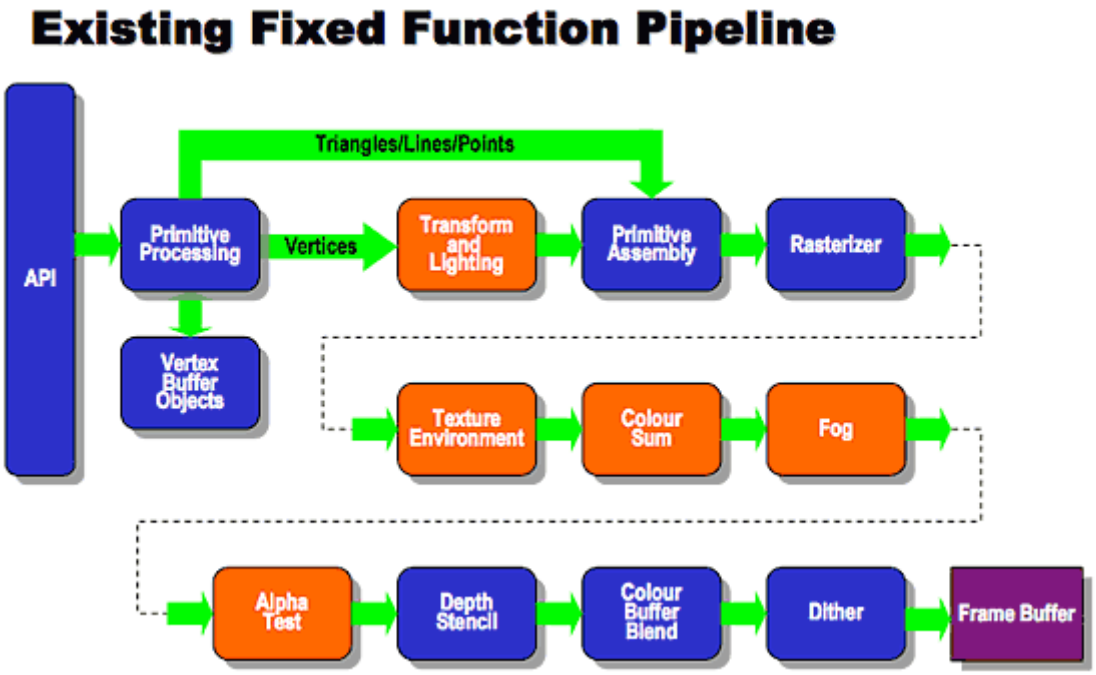
2 Moderni 3D-grafiikka

3D-grafiikan esittämiseen tietokoneella on useita toteutustapoja. Fotorealistisin lopputulos saadaan käyttämällä säteenseurantatekniikkaa (eng. ray tracing). Tämä tekniikka perustuu yksittäisten valonsäteiden simuloimiseen. Jokaista kuvan pikseliä kohti lasketaan yksi tai useampia valonsäteitä, jotka heijastetaan, sirotaan ym. valo-opin mukaan 3D-maailman geometriasta. Mitä enemmän valonsäteitä lasketaan, sitä fotorealistisempaan lopputulokseen päästään. Tekniikka on kuitenkin laskennallisesti äärimmäisen raskas, eikä täten sovellu sellaisenaan reaaliaikaisen 3D-grafiikkaan esittämiseen.

Ylivoimaisesti käytetyin tekniikka 3D-grafiikan esittämiseen on rasterointi. Siinä 3D-maailman muodot projisoidaan kuvitteellisen kameran linssille. Syntyneet ääriviivat rasteroidaan pikseleiksi ja näin saadaan karkea kuva 3D-maailman muodoista. Tekniikka ei ota kantaa pintojen värien laskemiseen, vaan tähän joudutaan käyttämään muita tekniikoita. Rasterointi on erittäin nopea tapa piirtää 3D-grafiikkaa, mutta menetelmän yksinkertaisuuden vuoksi sillä on vaikea päästä fotorealistiseen lopputulokseen. Tekniikan ylivoimaisen nopeuden vuoksi se on kuitenkin saavuttanut horjumattoman aseman reaaliaikaisen 3D-grafiikan esittämisessä. Modernit 3D-näytönohjaimet on varta vasten suunniteltu kykenemään rasteroimaan miljoonia polygoneja sekunnissa. Seuraavassa tutustumme modernin grafiikkasuorittimen rakenteeseen ja toimintaan.

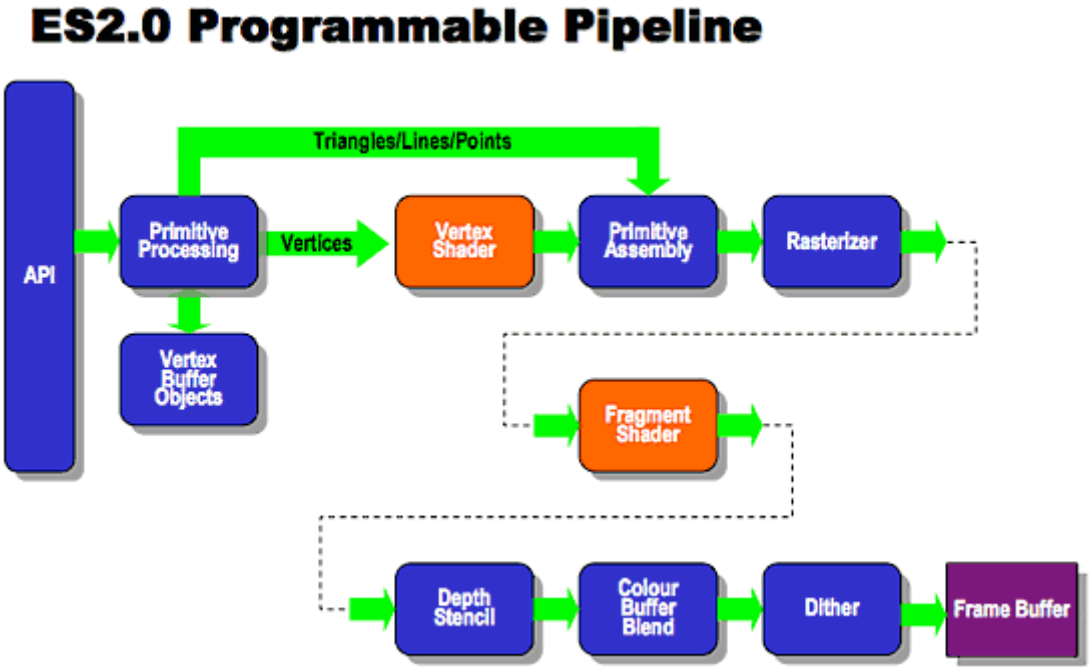
Grafiikkasuoritin eli näytönohjain on tietokoneen komponentti, jonka tehtävänä on muodostaa ja piirtää kuva näytölle. Jo parinkymmenen vuoden ajan näytönohjaimissa on ollut tuki 3D-grafiikan prosessoinnille ja piirtämiselle. Näytönohjain vastaanottaa ohjelman lähettämät 3D-geometrian kulmapisteet, projisoi nämä ja rasteroi lopullisen kuvan. Kaikki nämä vaiheet jakautuvat lukuisiin alivaiheisiin. Kuvassa 1 on esitettynä 3D-näytönohjaimen ”pipeline”, eli vaiheet, jotka kulmapisteet käyvät läpi päätyäkseen lopulta pikseleiksi näytöllä. Kuvassa esitetty prosessi on tätä nykyä jo vanhentunut ns. fixed-function pipeline (FFP). Tätä rakennetta käyttävien näytönohjainten toiminta oli varsin kiinteästi määrättyä. Ohjelmoija pystyi vaikuttamaan kulmapisteiden prosessointiin ja kuvan piirtymiseen ainoastaan asettamalla näytönohjaimelle tiettyjä

ennalta määrättyjä asetuksia. Todellista ohjelmoitavuutta ei FFP-näytönohjaimella voinut saavuttaa, ja tämän takia ennen varjostinohjelmien kehittämistä tehdyt 3D-pelit olivat graafisesti varsin vaatimattomia.



Kuva 1. FFP-grafiikkasuorittimen rakenne [13]

Vuoden 2002 tienoilla julkaistiin ensimmäiset varjostinohjelmia tukevat näytönohjaimet. Varjostinohjelma (eng. shader) on lyhyt ohjelmanpätkä, jonka ohjelmoija kirjoittaa ja antaa näytönohjaimelle suoritettavaksi. Varjostinohjelmat korvaavat suuren osan FFP-näytönohjainten konfiguroitavista osuuksista, ja näin mahdollistavat mielivaltaisen monimutkaisten graafisten efektien toteuttamisen. Kuvassa 2 on esitetty varjostinohjelmia tukevan näytönohjaimen rakenne. Kuvassa esiintyvät kaksi varjostinohjelmatyyppiä ovat verteksivarjostinohjelma (vertex shader) ja pikselivarjostinohjelma (fragment shader). Ensimmäiseksi mainittu hoitaa kulmapisteiden prosessoinnin, ja jälkimmäisellä voidaan vaikuttaa näytölle päätyvien pikselien väriin. Vuosien varrella varjostinohjelmien tukemien ominaisuuksien määrä on kasvanut samalla näytönohjainten laskutehon kasvaessa, ja nykyään varjostinohjelmien avulla voidaankin toteuttaa häkellyttävän realistista grafiikkaa, joka vetää vertoja jopa säteenseurantatekniikalla tuotetulle grafiikalle.



Kuva 2. Varjostinohjelmapohjaisen grafiikkasuorittimen rakenne [14]

2.1 3D-grafiikan periaatteet

Tässä luvussa käydään lyhyesti läpi modernin reaaliaikaisen 3D-grafiikan toiminen. Läpikäytävät periaatteet pätevät nimenomaan rasterointitekniikkaan, vaikkakin alkupään vaiheita käytetään myös ray tracing -tekniikassa. Läpikäytävät vaiheet on löydettävissä myös kuvan 2 tapahtumaketjusta, ja kuvaan viitataankin selitettäessä eri vaiheita.

Pohjimmiltaan 3D-grafiikka on puhdasta matematiikkaa. Esitettävät 3D-kappaleet ovat tietokoneen muistissa kulmapisteittensä avulla ilmaistuina. Kulmapisteet, joista käytetään usein nimitystä verteksi (eng. *vertex*), koostuvat kyseisen pisteen x, y ja z-koordinaateista. Usein mukana on myös kyseisen pisteen pinnan normaalivektorin ilmaisevat koordinaatit sekä tekstuurikoordinaatit, joista lisää myöhemmissä kappaleissa. Verteksit välitetään grafiikkasuorittimelle, joka aloittaa niiden prosessoinnin (kuvan 2 vaihe ”vertex shader”). Verteksiprosessoinnin tavoitteena on muuntaa verteksit 3D-kappaleen paikallisesta koordinaatistosta näytön kaksiulotteiseen koordinaatistoon. Tämä muunnos on käytännössä pelkkä matriisikertolasku. Verteksiprosessoinnille tulee välittää 4x4-kokoinen *muunnosmatriisi*, joka on varta

vasten laskettu siten, että kyseinen muunnos tapahtuu halutulla tavalla. Usein tämä matriisi on jaettu kolmeen erilliseen matriisiin ohjelmoinnin helpottamiseksi: *maailmamatriisi* (eng. *world matrix*) muuntaa verteksit kappaleen paikallisesta koordinaatistosta 3D-maailman globaaliin koordinaatistoon, kameran maailmamatriisin käänteismatriisi (eng. *view matrix)* muuntaa globaalissa koordinaatistossa olevat verteksit kameran paikalliseen koordinaatistoon, ja viimeisenä projektiomatriisi (eng. *projection matrix*) muuntaa verteksit kameran paikallisesta koordinaatistosta projisoituun koordinaatistoon. Haluttaessa realistinen projektio käytetään projektiomatriisina *perspektiivimatriisia*, jolla kertominen saa kappaleet pienenemään ja suurenemaan etäisyyden mukaan. Matriisimuunnoksen jälkeen vertekseille suoritetaan vielä muutama vaihe (jakaminen w-koordinaatilla ja skaalaaminen piirtopinnan koon mukaan), jonka jälkeen ne ovat näytön kaksiulotteisessa koordinaatistossa, jossa koordinaattiparit vastaavat suoraan näytön pikseleitä.

Verteksiprosessoinnin jälkeen grafiikkapiiri suorittaa varsinaisen rasteroinnin, eli laskee muunnettujen verteksien perusteella pikselit, jotka ovat verteksien rajaamien tahojen alueella. Rasterointivaiheeseen ei pysty vaikuttamaan ohjelmallisesti, vaan grafiikkasuoritin suorittaa tämän vaiheen aina samalla tavalla.

Rasteroinnin lopputuloksena on tiedossa pikselit, jotka näytön pinnalle projisoitu 3D-kappale vie. Näille pikseleille suoritetaan pikseliprosessointi, joka tätä nykyä tarkoittaa pikselivarjostinohjelman ajamista (kuvan 2 vaihe ”fragment shader”). Pikseliprosessoinnin seurauksena määräytyy pikselin lopullinen väri. Tässä vaiheessa mukaan astuvat mahdolliset tekstuurit, joilla 3D-kappale halutaan ”tapetoida”. Tekstuurit ovat bittikarttoja, jotka projisoidaan 3D-kappaleen pinnalle antaen sille sen ominaisen ulkonäön, oli kyse sitten puulaatikosta tai ihmiskasvoista. Bittikarttojen projisointi suoritetaan käyttämällä verteksiprosessointivaiheesta saatuja tekstuurikoordinaatteja (tunnetaan myös nimellä UV-koordinaatit). Tekstuurikoordinaatit ikään kuin lukitsevat bittikartan tietyn pisteen tiettyyn kappaleen verteksiin. Bittikartan *tekseli* (eng. *texel*, *texture pixel*) luetaan käyttäen tekstuurikoordinaattia, ja luettu väriarvo annetaan näytön pikselille.

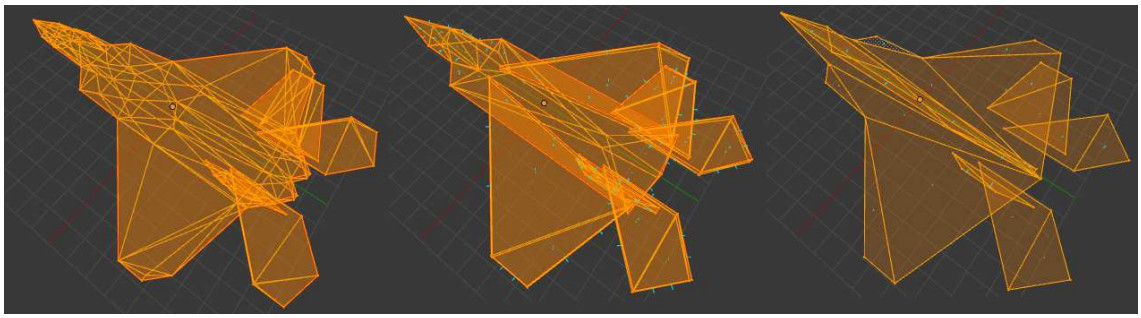
Pikseliprosessoinnin aikana suoritetaan yleensä myös valaistuksen laskeminen. Tähän tarvitaan pinnan normaalivektori, joka voidaan antaa joko verteksimäärityksen yhteydessä, tai sitten se voidaan lukea ns. normaalikartasta (eng. *normal map*), jolloin päästään huomattavasti tarkempaan lopputulokseen. Pinnan normaalin ja valojen

suuntavektorien avulla lasketaan pikselille kirkkausarvo, joka ilmaisee, paljonko valoa kyseisen pikselin ilmaisemaan kappaleen kohtaan osuu. Saadun arvon avulla tekstuureista luettuja väriarvoja voidaan moduloida, jolloin lopputuloksena on realistisesti valaistuksen perusteella kirkkauttaan vaihtava kappale. On huomionarvoista, että valaistus voidaan laskea myös verteksiprosessoinnin yhteydessä. Tämä on laskennallisesti huomattavasti kevyempää kuin valaistuksen pikselikohtainen laskeminen, mutta vaikuttaa myös negatiivisesti kuvanlaatuun.

2.2 Suorituskykyongelmia

3D-grafiikan reaaliaikaiseen esittämiseen liittyy huomattava määrä potentiaalisia suorituskykyongelmia. Moderneissa 3D-pelien maailmoissa voi olla kerralla näkyvillä miljoonia verteksejä, mikä tarkoittaa suoraan vähintään yhtä montaa kertaa verteksivarjostinohjelman suorittamista, ja useasti monin verroin pikselivarjostinohjelman suorituskertoja. Kun tähän lisätään useimpien pelaajien esittämä vaatimus vähintään 60 ruutua sekunnissa ruudunpäivitysnopeudesta, grafiikkasuorittimen käsittelemä data- ja komentomäärä nousee tähtitieteelliseksi. Tässä luvussa käydään läpi useimmiten ilmenevät pullonkaulakohdat ja esitellään lyhyesti niihin löytyvät yleisimmät ratkaisutavat.

Puhuttaessa grafiikkasuorittimen suorituskyvystä on eräs keskeisimmistä käsitteistä *triangle rate*, eli se kuinka monta kolmiota (tarkemmin sanottuna kolmion kulmapistettä eli verteksiä) grafiikkasuoritin kykenee käsittelemään sekunnissa. Kyseinen arvo on yleensä täysin teoreettinen maksimi, joka on saavutettavissa vain tarkoin suunnitelluissa testitilanteissa. Todellisessa tapauksessa vertekseille halutaan usein tehdä melko raskaitakin operaatioita verteksivarjostimessa, jolloin verteksien maksimimäärä alenee radikaalisti. Verteksimäärien alentamiseksi on olemassa paljon erilaisia tekniikoita. Yksinkertainen tapa on suorittaa piirrettäville kappaleille näkyvyystarkastus. Sen tarkoituksena on tunnistaa ja poistaa kameralta piilossa olevat kappaleet ennen kuin ne edes lähetetään grafiikkasuorittimelle. Toinen tehokas tapa on käyttää LOD-tekniikkaa (*level of detail*). 3D-kappaleista luodaan useita eri resoluutioisia versioita, ja piirrettävä versio valitaan kameran ja kappaleen välisen etäisyyden perusteella. Näin kaukana olevat kappaleet piirretään pienellä määrällä verteksejä, ja lähellä oleville kappaleille jää suuri määrä verteksejä käyttöön. Kuvassa 3 on esimerkkinä erään 3D-mallin kolme eri resoluutioista versiota.



Kuva 3. LOD-tekniikka

Toinen keskeinen suorituskykyyn liittyvä termi on *fill rate*, joka viittaa grafiikkasuorittimen kykenemään määrään prosessoida pikseleitä. Useimmiten juuri tämä arvo aiheuttaa ensimmäisenä suorituskykyongelmia johtuen käsiteltävien pikselien paljon suuremmasta määrästä suhteessa verteksien määrään. Esimerkiksi käytettäessä Full HD -resoluutiota (1920x1080) käsiteltäviä pikseleitä on jo yhdellä ainoalla ruudullisella yli 2 miljoonaa. Kun tämä kerrotaan ruudunpäivitysnopeudella ja otetaan huomioon *overdraw*, eli se, että 3D-maailman kappaleita piirtyy usein päällekkäin jolloin yhtä näytön pikseliä kohden joudutaan prosessoimaan useampi pikseli, ollaan erittäin suurissa lukemissa. Pikselimäärän vähentäminen onkin ensiarvoisen tärkeää, ja tähän on olemassa monenlaisia tekniikoita. Koko 3D-maailma voidaan esimerkiksi piirtää näytön resoluutiota alempiresoluutioiselle piirtopinnalle, ja skaalata se jälkikäteen näytön resoluutioon. Näin käsiteltävien pikselien määrä putoaa valtavasti, mutta samalla kuvanlaatu kärsii. Toinen mielenkiintoinen tekniikka on suorittaa ns. *z-prepass*, joka tarkoittaa 3D-maailman piirtämistä kahteen kertaan. Ensimmäisen piirtokerran ainoana tavoitteena on kirjoittaa pikselien syvyysarvot syvyyspuskuriin, ja se voidaan toteuttaa käyttämällä kevyitä yksinkertaisia varjostinohjelmia ilman valaistusta tai teksturointia. Jälkimmäinen piirtokerta suoritetaan käyttämällä normaaleja varjostinohjelmia, mutta koska syvyyspuskuri sisältää jo syvyysarvot, pystytään saman pikselin turha uudelleenkäsittely estämään, ja näin *overdraw*-ongelma saadaan täysin hävitettyä [7].

Kolmas ongelma, joka ei varsinaisesti liity itse grafiikkasuorittimen suorituskykyyn, liittyy grafiikkarajapinnan kautta lähetettyjen piirtokutsujen ja tilavaihdosten määrään. Haluttaessa piirtää mitä tahansa grafiikkasuorittimelle on lähetettävä tieto käytettävistä varjostinohjelmista, tekstuureista, verteksipuskureista sekä muista detaileista sekä tietenkin myös itse piirtokomento. Kaiken tämän tiedon siirtäminen vie aikaa, eikä grafiikkasuoritin voi aloittaa kuvan piirtämistä ennen kuin kaikki piirtokomennot on

lähetetty. Mikäli piirtokutsuja on suuri määrä, voi tästä muodostua pullonkaula grafiikkasuorittimen joutuessa odottamaan toimettomana piirtokomentojen lähetyksen valmistumista. Jotta näin ei kävisi, piirtokutsujen määrä on pyrittävä pitämään mahdollisimman pienenä. Tähän voidaan käyttää monia tekniikoita. Mikäli samaa 3D-kappaletta piirretään useita kertoja, voidaan nämä yhdistää yhdeksi piirtokutsuksi monistamalla kappaleiden verteksit. Haittapuolena on muistinkulutuksen kasvu, eikä tekniikka sovellu dynaamisten kappaleiden piirtämiseen. Mikäli käytössä on tarpeeksi moderni grafiikkasuoritin, voidaan käyttää ns. instansoitua piirtämistä [15]. Tätä tekniikkaa käytettäessä monistettavan kappaleen verteksit lähetetään vain kerran, minkä lisäksi lähetetään lista instansoitavien kappaleiden maailmamatriiseja, joista verteksivarjostimessa voidaan valita oikea instanssitunnisteen avulla.

3 OpenGL-rajapinta

OpenGL on avoin standardi, joka määrittelee ohjelmointirajapinnan grafiikkasuorittimen ja ohjelmien välillä [16]. Rajapinta tarjoaa menetelmät sekä 2D- että 3D-grafiikan piirtämiseen. Koska kyseessä on avoin standardi, on rajapinnalle erittäin laaja laitetuki. Lähes kaikki 3D-grafiikkaan kykenevät laitteet tukevat OpenGL:ää, mukaan lukien iOS- ja Android-laitteet. Melkeinpä ainoan poikkeuksen tekee Microsoftin pelikonsolit ja mobiililaitteet, jotka käyttävät Microsoftin omaa DirectX-rajapintaa.

OpenGL-standardia kehittää tätä nykyä Khronos Group -niminen taho, joka koostuu grafiikkasuorittimia ja niiden ohjelmistoja kehittävistä yhtiöistä. OpenGL-standardi seuraakin hyvin läheisesti grafiikkasuorittimien kehittymistä, ja uusimmat ominaisuudet ovat OpenGL:n käytettävissä useasti ennen kilpailevaa DirectX-standardia.

OpenGL:n alkuaikoina rajapinta oli huomattavasti laajempi ja monimutkaisempi kuin nykyään. Tämä johtui siitä, että alun perin OpenGL oli suunniteltu lähinnä CAD-ohjelmistojen kehittämiseen, ja suuri osa rajapinnan funktioista toteutti tällaisissa ohjelmissa tarvittavia erityistoimintoja [12]. Myös grafiikkasuorittimien rakenne poikkesi huomattavasti nykyisestä. 2000-luvun alkuun saakka grafiikkasuorittimet toimivat ns. FFP-periaatteella, joka poikkeaa huomattavasti nykyään käytettävästä varjostinohjelmapohjaisesta toiminnasta.

Modernien OpenGL-versioiden rajapinta on tarkoin rajattu kattamaan vain ne ominaisuudet, joita nykyiset grafiikkasuorittimet tukevat. Kaikki ns. legacy-toiminnallisuus on poistettu, mukaan lukien tuki FFP:lle. OpenGL:n käyttö nykyään onkin hyvin pitkälti varjostinohjelmien kirjoittamista.

3.1 OpenGL ES

OpenGL:stä on olemassa erityisesti sulautetuille järjestelmille tarkoitettu OpenGL ES -versio (OpenGL for Embedded Systems) [3]. Tämä versio on suunniteltu sulautettujen järjestelmien suorituskyky- ja virrankäyttörajoitteita ajatellen. Standardista on poistettu ominaisuuksia, joiden ei ole katsottu olevan tarpeellisia sulautetuissa järjestelmissä, sekä mm. lisätty tuki fixed-point-luvuille laitteille, joista puuttuu liukulukuprosessori. Käytännössä kaikki OpenGL:ää tukevat mobiililaitteet tukevat nimenomaan OpenGL ES -standardia.

OpenGL ES -standardin ensimmäiset versiot (1.0 ja 1.1) tukivat FFP:tä, mutta eivät varjostinohjelmia. Nämä versiot ovat kaikkein laajimmin tuettuina mobiililaitteissa. Androidissa on ollut tuki näille versiosta 1.6 asti, ja myös ensimmäisen sukupolven iOS-laitteet tukevat näitä [3].

OpenGL ES -standardin versio 2.0 on jo huomattavasti kehittyneempi ominaisuuksiltaan. FFP-tuki on korvattu varjostinohjelmilla, aivan kuten uusimmissa OpenGL:n täysversioissakin. Laitetuki tälle versiolle on varsin laaja: yli 95 % Android-laitteista tukee sitä, ja myös kaikki markkinoilla olevat iOS-laitteet tukevat sitä [1, 2]. OpenGL ES 2.0 on hyvä kompromissi laitetuen laajuuden sekä rajapinnan tukemien ominaisuuksien välillä, ja senpä takia se on valittu käytettäväksi tässä insinöörityössä.

Uusimpana tulokkaana on OpenGL ES -standardin vuonna 2012 julkaistu 3.0-versio. Tämä versio laajentaa tuettujen ominaisuuksien kirjoa entisestään lisäten tuen mm. instansoidulle piirtämiselle, sekä kauan kaivatun tuen standardoidulle pakatulle tekstuuriformaatille. Johtuen version tuoreudesta laitetuki on vielä hyvin minimaalinen. Joulukuussa 2013 vain 3,6 % Android-laitteista tuki sitä, ja iOS-puolella ainoa tuettu laite oli iPhone 5S [1, 2].

3.2 OpenGL ES 2.0:n käyttäminen

OpenGL ES 2.0:n käyttäminen on rajapinnan minimalistisuuden johdosta varsin suoraviivaista. Piirtäminen rajapinnalla käsittää seuraavat vaiheet:

- Luodaan OpenGL-konteksti
- Ladataan käytettävät piirtoresurssit (tekstuurit, verteksipuskurit ja varjostinohjelmat) grafiikkasuorittimen muistiin
- Otetaan käyttöön halutut piirtoresurssit
- Annetaan piirtokomento

OpenGL-konteksti luodaan käyttämällä käyttöjärjestelmän tarjoamaa rajapintaa. Windowsilla tähän tarkoitukseen on WGL-kirjasto, Linuxissa GLX-kirjasto ja Androidilla EGL-kirjasto. Kirjaston avulla luodaan myös yleensä samalla piirtoikkuna, jonka asetukset sopivat laitteistokiihdytettyyn piirtämiseen. Androidilla kaikki tämä voidaan ohittaa käyttämällä Android API:n GLSurfaceView-ikkunaelementtiä, joka hoitaa sisäisesti OpenGL-kontekstin luomisen ja tuhoamisen.

Piirtoresurssien luomiseen käytetään OpenGL:n *glGen*-alkuisia funktioita. Nämä funktiot palauttavat tunnisteen luotuun resurssiin. Resurssi voidaan tämän tunnisteen avulla ns. “bindata”, eli ottaa käyttöön *glBind*-alkuisilla funktioilla. Käyttöönotettuun resurssiin voidaan ladata varsinainen data käyttämällä datan lataamiseen tarkoitettuja funktioita. Verteksilistojen tapauksessa tarvittava funktio on *glBufferData*, ja tekstuurien tapauksessa *glTexImage2D*.

Varjostinohjelmien luominen ja käyttöönottaminen on hieman hankalampi operaatio. Verteksi- ja pikselivarjostinohjelmien GLESSL-kieliset lähdekoodit (ks. luku 3.3) on ensin käännettävä grafiikkasuorittimen ymmärtämään binäärimuotoon käyttämällä *glShaderSource* ja *glCompileShader* -funktioita. Tämän jälkeen käännetyt binäärikoodit yhdistetään yhdeksi objektiksi kutsumalla *glCreateProgram*, *glAttachShader* ja *glLinkProgram* -funktioita.

Kun halutaan piirtää tiettyjä piirtoresursseja käyttäen, on halutut resurssit otettava ensin käyttöön *glBind*-funktioilla, aivan kuten resurssien luomisvaiheessakin. Haluttu varjostinohjelmaobjekti (sisältää sekä verteksi- että pikselivarjostinohjelman) otetaan

käyttöön *glUseProgram*-funktiolla. Tämän jälkeen annetaan itse piirtokomento. Käytettäessä indeksoimatonta piirtoa käytetään *glDrawArrays*-funktiota, ja indeksoitu piirto hoidetaan *glDrawElements*-funktiolla.

3.3 GLSL

GLSL eli OpenGL Shading Language on ohjelmointikieli, jolla OpenGL:n käyttämät varjostinohjelmat kirjoitetaan [17]. Kielen avulla voidaan kirjoittaa kaikkia näytönohjainten tukemia varjostinohjelmatyyppejä (verteksi-, pikseli-, geometria- ja tesselaatiovarjostinohjelmat). Kieli on syntaksiltaan lähellä C-kieltä, mutta tarjoaa korkeamman tason ominaisuuksia, kuten esimerkiksi matemaattisten operaattoreiden ylikirjoituksen vektoreille ja matriiseille. Listauksessa 1 on esimerkki GLSL:llä kirjoitetusta yksinkertaisesta verteksivarjostinohjelmasta.

Kuten OpenGL-rajapinnasta, myös GLSL:stä on olemassa sulautetuilla laitteille tarkoitettu erityisversio. GLESSL:n (OpenGL ES Shading Language) ominaisuuskirjoa on karsittu, ja kielellä ei esimerkiksi voi lainkaan kirjoittaa geometria- ja tesselaatiovarjostinohjelmia. Versioon on myös lisätty tuki tarkkuusmäärityksille eri tietotyypeille, joiden avulla varjostinohjelmien suoritusnopeutta voidaan parantaa.

```
uniform mat4 uViewProjMatrix;
uniform mat4 uWorldMatrix;
attribute vec3 aPosition;

void main(void)
{
          gl_Position = uViewProjMatrix * uWorldMatrix *
                                vec4(aPosition, 1.0);
}
```

Koodiesimerkki 1. GLSL-verteksivarjostinohjelma

4 Grafiikkamoottoreista

Termi grafiikkamoottori ei ole kovin tarkoin määritelty. Useasti puhuttaessa pelien tekniikasta grafiikkamoottorin katsotaan olevan kiinteä osa pelimoottoria, eikä sitä käsitellä omana komponenttinaan. On kuitenkin olemassa myös lukuisia ohjelmia, joita

kutsutaan nimenomaan grafiikkamoottoreiksi eikä pelimoottoreiksi. Mikäli termille olisi kehitettävä määritelmä, voisi se olla seuraavanlainen: grafiikkamoottori on ohjelmakomponentti, joka tarjoaa käyttäjälleen yksinkertaistetun tavan esittää 2D- tai 3D-gafiikkaa, piilottaen grafiikkarajapintojen yksityiskohdat korkeamman tason käsitteiden taakse. Grafiikkamoottorin käyttämiseksi ei tarvitse luoda ja käsitellä itse matalan tason piirtoresursseja, kuten verteksipuskureita tai tekstuureja, vaan grafiikkamoottori tarjoaa yleensä jonkinlaisen piirtokappaleolion, joka hoitaa sisäisesti nämä asiat. Käyttäjän tehtäväksi jää luoda ja liikutella kappaleita. Toki useasti grafiikkamoottorit sallivat osaavan käyttäjän päästä käsiksi myös matalan tason osiinsa, jotta moottorin toimintaa voidaan muokata tarkemmin. Tässäkin tapauksessa grafiikkamoottoria käyttämällä saavutetaan hyötyjä suhteessa suoraan grafiikkarajapintojen käyttämiseen, koska usein grafiikkaamoottori on abstrahoinut myös matalan tason käsitteet omien rajapintojensa taakse. Näin grafiikkamoottori voi tukea esimerkiksi sekä OpenGL- että Direct3D-grafiikkarajapintoja.

4.1 Valmiita grafiikka- ja pelimoottoreita

Tässä luvussa käydään läpi muutamia suosituimpia peli- ja grafiikkamoottoreita. Moottorien käyttötarkoitus ja ominaisuudet käydään pintapuolisesti läpi. Tarkoituksena ei ole suorittaa todellista vertailevaa tutkimusta, vaan lähinnä saada karkea käsitys siitä, minkälaisia tuotteita markkinoilla jo on.

Unity

Unity on yksi tämän hetken suosituimmista pelimoottoreista [18]. Se on saavuttanut valtaisan suosion etenkin indie-pelikehittäjien parissa johtuen suotuisasta lisenssistään sekä helposta omaksuttavuudestaan. Unity on kokonainen pelimoottori, eli sisältää grafiikkamoottorin lisäksi kaiken tarvittavan valmiin pelin tuottamiseksi. Unityllä tehdyt pelit voidaan kääntää myös mobiililaitteille.

Unityn grafiikkamoottori tarjoaa valtavan ominaisuuskirjon. Tarjolla on monia valmiita valaistusmenetelmiä sekä 100 valmista varjostinohjelmaa [10]. Kuten muutkin pelin osa-alueet, myös grafiikkaa voidaan muokata hyvin pitkälle Unityn graafisella kehitysympäristöllä. Graafinen editori onkin yksi selkeä syy Unityn suurelle suosiolle etenkin aloittelevien pelikehittäjien keskuudessa. Unityn eräänä ongelmana voidaan

pitää sitä, ettei lähdekoodiin ole pääsyä. Kaikki ohjelmointi hoidetaan korkean tason kielillä (käytettävissä ovat JavaScript, C# ja Boo), mistä voi seurata suorituskykyongelmia haluttaessa implementoida jokin uusi paljon laskentatehoa vaativa ominaisuus.

Ogre

Ogre on avoimen lähdekoodin grafiikkamoottori [19]. Se keskittyy pelkkään grafiikkaan, eikä sisällä muita peleissä vaadittavia komponentteja, joskin joitain lisäominaisuuksia on saatavilla käyttäjien tekemien lisäosien kautta. Ogren rajapinta on suunniteltu olio-ohjelmointilähtöisesti, ja sen käyttö vaatii vahvaa tuntemusta C++ -kielestä. Tuetuilta ominaisuuksiltaan Ogre on varsin hyvin varusteltu. Se ei tarjoa juurikaan valmiita varjostinohjelmia, mutta sen rajapintojen kautta omien varjostinohjelmien integroiminen on helppoa, ja ylipäätään lähes kaikkia moottorin ominaisuuksia on mahdollista muokata. Muita mainittavia ominaisuuksia ovat tuki luurankoanimaatioille ja hierarkkinen objektien hallinta [11]. Valitettavasti tukea mobiilialustoille ei Ogresta löydy, vaikkakin epävirallisia keskeneräisiä käännösyrityksiä on olemassa [20].

Irrlicht Engine

Irrlicht Engine on toinen suosittu grafiikkamoottori [21]. Ominaisuuksiensa puolesta se on varsin lähellä Ogrea [22], mutta sen käyttäminen on jonkin verran yksinkertaisempaa. Yleisesti ottaen Irrlicht Engine ei salli yhtä vapaamuotoista muokkaamista kuin Ogre, vaikka toki avointen lähdekoodiensa ansiosta siihenkin voidaan kirjoittaa omia lisäosia. Myöskään Irrlicht Enginestä ei ole olemassa virallista mobiiliversiota.

4.2 Grafiikkamoottorin tekemisen motiivit

Toimivan grafiikkamoottorin kehittäminen on varsin suuri haaste. Se vaatii vahvaa tuntemusta grafiikkarajapintojen toiminnasta, sekä tietenkin 3D-grafiikan ja siihen liittyvien matemaattisten käsitteiden ymmärrystä. Mikäli grafiikkamoottorin halutaan vieläpä toimivan nopeasti ja käyttävän vähän resursseja, vaaditaan ymmärrystä optimoinnista ja grafiikkasuoritinten toiminnasta. Oman grafiikkamoottorin kehittämisen motiiveina toimivatkin lähinnä halu oppia lisää kyseisistä asioista.

Syy, miksi grafiikkamoottori päätettiin tehdä juuri mobiililaitteita ajatellen, on kaksiosainen. Mobiililaitteille julkaistut 3D-pelit ovat yleisesti ottaen graafisesti vähemmän näyttäviä kuin vastaavat PC-pelit. Näin siis grafiikkamoottorilla voidaan tuottaa jotenkuten kelvollisen näköisiä pelejä, vaikkei se tukisikaan kaikkia uusimpia graafisia efektejä. Toisaalta mobiililaitteiden resurssit ovat huomattavasti rajallisemmat kuin PC-koneiden, joten tässä mielessä grafiikkamoottorin kehittäminen mobiililaitteille tarjoaa erityisiä haasteita optimoinnin saralla.

5 Määrittely

Tässä luvussa kuvataan grafiikkamoottorin korkean tason vaatimusten määrittelyä. Esiin tuodaan grafiikkamoottorin toteuttamisen ensisijaiset tavoitteet sekä ensimmäisen version tukemat ominaisuudet. Myös grafiikkamoottorin minimilaitteistovaatimukset määritetään.

5.1 Tavoite

Tavoitteena oli kehittää yksinkertainen 3D-grafiikkamoottori, jota voidaan käyttää pelien ja muiden 3D-grafiikkaa vaativien sovellusten kehittämiseen. Yksinkertaisuus tässä kontekstissa tarkoittaa lähinnä sitä, että moottori tarjoaa varsin matalan tason rajapinnan käyttäjälleen. Moottori abstrahoi OpenGL:n käyttämät käsitteet, kuten verteksipuskurit ja varjostinohjelmat, omien rajapintojensa taakse ja mahdollistaa näin 3D-grafiikan käyttämisen, vaikkei OpenGL:ää tuntisikaan. Moottorin ei ole tarkoitus tarjota monimutkaisia korkean tason ominaisuuksia, joita on saatavilla monissa tunnetuissa grafiikkamoottoreissa, kuten esimerkiksi animaatiojärjestelmiä ja valmiita varjostinohjelmaefektejä.

Grafiikkamoottorin päätavoitteina oli

- rakenteellinen selkeys ja helppo laajennettavuus
- toimivuus mobiililaitteilla
- kohtuullinen suorituskyky.

Eräs grafiikkamoottorin toteuttamisen päämotiiveista oli tutustuttaa tekijänsä syvemmin 3D-grafiikkaan sekä erityisesti OpenGL-rajapintaan, joten moottorin toteutuksessa pyrittiin selkeyteen ja opettavaisuuteen. Moottorin rakenteen suunnittelussa pyrittiin käyttämään hyväksi havaittuja olio-ohjelmoinnin periaatteita jakamalla moottori loogisiin osiin. Moottorin ensimmäisen version tukemat ominaisuudet rajattiin tarkasti, joten moottorista pyrittiin tekemään mahdollisimman modulaarinen mahdollista myöhempää jatkokehitystä varten.

Grafiikkamoottorissa käytetään OpenGL ES 2.0 -rajapintaa, joka on tuettuna hyvin laajasti mobiililaitteilla sekä myös työpöytäpuolella. Tämän takia moottori toimii teoriassa melkein millä tahansa laitteella, ja tarkoituksena olikin tehdä siitä hyvin pitkälti alustariippumaton. Tämä toteutettiin piilottamalla alustariippuvaiset osat moottorin alimpiin kerroksiin, jolloin ainoastaan nämä joudutaan kirjoittamaan uudestaan siirrettäessä moottoria uudelle alustalle.

Vaikka ensisijainen tavoite ei ollutkaan grafiikkamoottorin maksimaalinen suorituskyky, kiinnitettiin suunnittelussa silti huomiota keskeisimpiin 3D-grafiikan pullonkaulakohtiin ja pyrittiin löytämään kompromissi rakenteen selkeyden ja optimoinnin välillä. Grafiikkamoottorin suorituskyvyn kannalta keskeisimpien rakenteiden suunnittelussa seurattiin menetelmiä, joita käytetään myös kaupallisissa grafiikkamoottoreissa.

5.2 Tuetut ominaisuudet

Grafiikkamoottorin ensimmäisen version tukemat ominaisuudet ovat hyvin rajalliset. Moottorilla pystyy lähinnä lataamaan ja piirtämään staattisia 3D-malleja varjostinohjelmia käyttäen. Kaikki monimutkaisemmat ominaisuudet puuttuvat. Tukea ei ole esimerkiksi animaatioille, eikä moottorissa ole valmista valaistusjärjestelmää. Valaistus, kuten muutkin monimutkaisemmat efektit, voidaan toteuttaa kirjoittamalla näitä tukeva varjostinohjelma, ja välittämällä valaistusparametrit varjostinohjelmalle parametreina.

Grafiikkamoottorissa ei tällä hetkellä ole myöskään ns. *scenegraph*-tyyppistä rakennetta, jolla piirrettäviä 3D-malleja voisi ryhmitellä hierarkkisesti, vaan moottorin käyttäjän on pidettävä itse kirjaa 3D-malliensa asettelusta. Tästä syystä moottorissa ei myöskään ole näkymättömissä olevien 3D-mallien automaattista poistoa, vaan

grafiikkamoottori piirtää aina kaiken, mitä käsketään. Näkymähierarkia, kuten myös näkymättömien kappaleiden poisto, on tarkoitus toteuttaa grafiikkamoottoria käyttävän ohjelman korkeammalla tasolla.

5.3 Laitteistovaatimukset

Grafiikkamoottori on tarkoitettu toimivaksi kaikilla OpenGL ES 2.0 -rajapintaa tukevilla laitteilla. Tarkemmat laitevaatimukset määräytyvät lähinnä sen perusteella, miten monimutkaisia 3D-maailmoja grafiikkamoottorilla haluaa esittää. Moottoria on kehitetty ja testattu Android-älypuhelimella, jossa on 800 MHz prosessori, 512 Mt keskusmuistia ja Adreno 205 –grafiikkasuoritin. Näitä voitaneen pitää jonkinasteisina minimivaatimuksina, mikä johtuu jo pelkästään siitä, ettei tätä huonompitehoisia laitteita juurikaan ole enää markkinoilla.

6 Suunnittelu ja toteutus

Määrittelyn pohjalta ruvettiin suunnittelemaan grafiikkamoottorin teknistä toteutusta. Tässä luvussa käsitellään moottorin suunnittelua ja teknisiä ratkaisuja, joihin suunnittelun perusteella päädyttiin.

6.1 Käytetyt tekniikat

Tässä luvussa esitellään lyhyesti grafiikkamoottorin toteutuksessa käytettyjä tekniikoita ja kirjastoja.

6.1.1 C++

C++ on eräs kaikkien aikojen käytetyimmistä ohjelmointikielistä. Se on pääasiallisesti oliopohjainen kieli, mutta sallii myös proseduraalisen ohjelmoinnin johtuen C-pohjaisuudestaan. C++:lla kirjoitetut ohjelmat käännetään kohdearkkitehtuurin konekielelle, mikä mahdollistaa ylivoimaisen suoritusnopeuden verrattuna tulkattuihin ja virtuaalikonepohjaisiin kieliin. C++:n avulla on mahdollista päästä käsiksi hyvin matalan tason ominaisuuksiin, kuten esimerkiksi monien nykyisten prosessoriarkkitehtuurien

tarjoamiin SIMD-käskykantoihin. Näistä on hyötyä erityisesti peli- ja multimediaohjelmoinnissa.

Varjopuolena C++:n tarjoamaan laajaan ominaisuuskirjoon kielen syntaksi on varsin monimutkainen. Etenkin kielen template-ominaisuudella on mahdollista luoda erittäin monimutkaisia rakenteita. Kielen tarjoama suorituskyky ei myöskään ole itsestäänselvyys, vaan ohjelmoijan on todella tiedettävä, mitä tekee saadakseen tehoja irti. Manuaalinen muistinhallinta pakottaa suunnittelemaan ohjelman tietojen käsittelyn huomattavasti tarkemmin kuin korkeamman tason kieliä käytettäessä.

C++:lla on horjumaton asema tarkasteltaessa markkinoilla olevia grafiikka- ja pelimoottoreita. Käytännössä kaikki suosituimmat pelimoottorit ovat C++:lla kirjoitettuja, mukaan lukien Unreal-enginen kaikki versiot, Unity ja Ogre. Monissa näissä moottoreissa korkean tason ominaisuuksia voidaan kirjoittaa erilaisilla skriptikielillä, mutta itse moottorin runko ja suorituskykyä vaativat osat ovat C++:aa.

Myös tässä työssä toteutettava grafiikkamoottori päätettiin kehittää C++:lla. Valintaperusteina painavat lähinnä edellä mainittu suorituskykyaspekti, sekä se, että kieli oli ennestään tuttu. C++ mahdollistaa myös moottorin helpon kääntämisen eri mobiilialustoille, mikäli tämä jossain vaiheessa tulee ajankohtaiseksi. Kaikki kolme suurinta mobiilialustaa (Android, iOS ja Windows Phone 8) tukevat natiivisovelluksia, jolloin ainoastaan pieni osa moottoria joudutaan kirjoittamaan uudestaan eri alustoille.

6.1.2 Android NDK

Android NDK (Native Development Kit) on Googlen ylläpitämä kokoelma kääntäjiä ja muita työkaluja, joiden avulla Android-käyttöjärjestelmälle voidaan kehittää natiivisovelluksia [23]. Kielenä sovelluksissa voidaan käyttää C:tä ja C++:aa. Android NDK tarjoaa myös kirjastot monien matalan tason ohjelmointirajapintojen käyttämiseen sovelluksissa, mukaan lukien OpenGL ES -kirjastot.

Android NDK:n avulla C/C++ -kielisestä lähdekoodista käännetään kirjastotiedosto mobiiliprosessorien arkkitehtuurille. Tällä hetkellä tuettuina ovat ARMv7 ja v6, MIPS ja x86-arkkitehtuurit. Kirjastotiedosto linkitetään mukaan muuhun sovellukseen, jossa voidaan kutsua kirjaston tarjoamia natiivifunktioita. Android NDK:ta ei siis ole tarkoitettu kokonaisen Android-sovelluksen kehittämiseen. Tarkoituksena on, että suurinta

mahdollista suorituskykyä kaipaavat sovellusosat kirjoitetaan natiivikoodina, ja loput sovelluksesta kehitetään normaalisti Javalla käyttäen Android SDK-työkalupakettia. Tosin Android-version 2.3 myötä Android NDK:lla on voinut kehittää myös pelkkää natiivikoodia sisältäviä täysin toimivia sovelluksia. Tällaiset sovellukset jäisivät kuitenkin hyvin pelkistetyiksi, koska Android NDK tarjoaa vain murto-osan SDK:n sisältämistä rajapinnoista. Grafiikkamoottorin minivaatimukseksi määritettiin Android-versio 2.2, joten sen kehittämisessä on jouduttu käyttämään myös hieman Javaa ja Android SDK:ta.

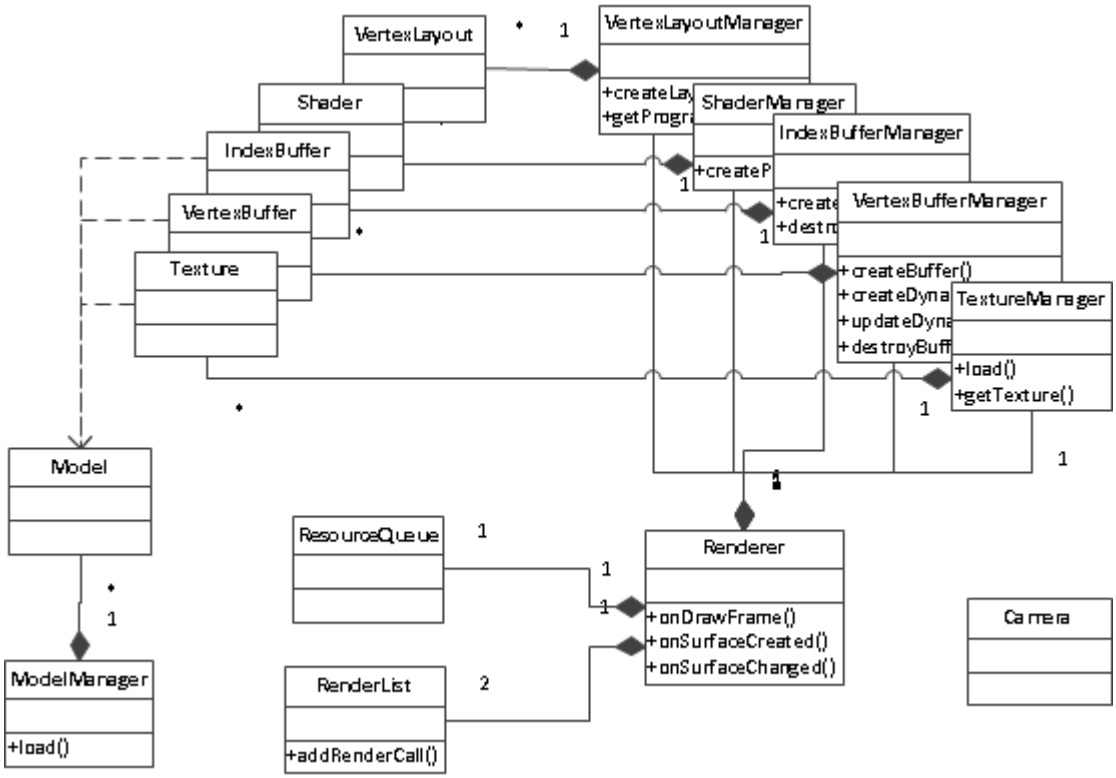
6.2 Yleinen rakenne

Grafiikkamoottorin rakennetta lähdettiin suunnittelemaan top-down-tyyppisesti. Ensimmäisenä määritettiin grafiikkamoottorin rajapinta, jolla se tulisi integroitumaan sitä käyttävään ohjelmaan. Koska moottori on ensisijaisesti Android-laitteille suunnattu, suunniteltiin tämä rajapinta olemaan mahdollisimman helposti kytkettävissä Androidin OpenGL-toteutukseen. Android tarjoaa OpenGL:n käyttämiseen valmiin käyttöliittymäkomponentin (GLSurfaceView), joka hoitaa sisäisesti OpenGL-kontekstin luomisen sekä käynnistää erillisen piirtosäikeen. Tämä oli luonnollinen lähtökohta grafiikkamoottorin integroimiseen. Grafiikkamoottori tarjoaa kolme julkista funktiota, *onDrawFrame, onSurfaceCreated* ja *onSurfaceChanged*, joita on tarkoitus kutsua GLSurfaceView'n vastaavissa callback-metodeissa. Metodia *onDrawFrame* kutsutaan, kun halutaan piirtää uusi ruudullinen, metodia *onSurfaceCreated* kutsutaan, kun piirtopinta ja OpenGL-konteksti on luotu ja metodia *onSurfaceChanged* kutsutaan, kun piirtopinnan koko muuttuu.

Alustarajapinnan suunnittelun jälkeen alettiin suunnitella grafiikkamoottorin sisäistä rakennetta. Alusta asti oli selvää, että grafiikkamoottori halutaan jakaa kahteen säikeeseen (luku 6.3), joten myös piirtokomentojono-rakenne (luku 6.4) oli selkeä valinta datan siirtämiseksi säikeiden välillä. Kuvassa 4 on esitettynä grafiikkamoottorin luokkakaavio. Piirtokomennot lisätään *RenderList*-luokkaan, joka välitetään *Renderer*-luokalle toiseen säikeeseen, missä varsinainen piirtäminen tapahtuu.

Luokkakaaviossa esiintyvät myös eri resurssityypejä käsittelevät manager-luokat. Kaikille tärkeimmille piirtoresursseille (tekstuurit, varjostinohjelmat ja verteksi- ja indeksipuskurit) luotiin omat manager-luokat, jotka hoitavat kyseisten resurssien

luomisen ja tuhoamisen. Resurssien hallinnassa käytetään *reference counting* -tekniikkaa, mikä mahdollistaa helpon kirjanpidon siitä, milloin mikäkin resurssi on tuhottava.



Kuva 4. Grafiikkamoottorin luokkakaavio

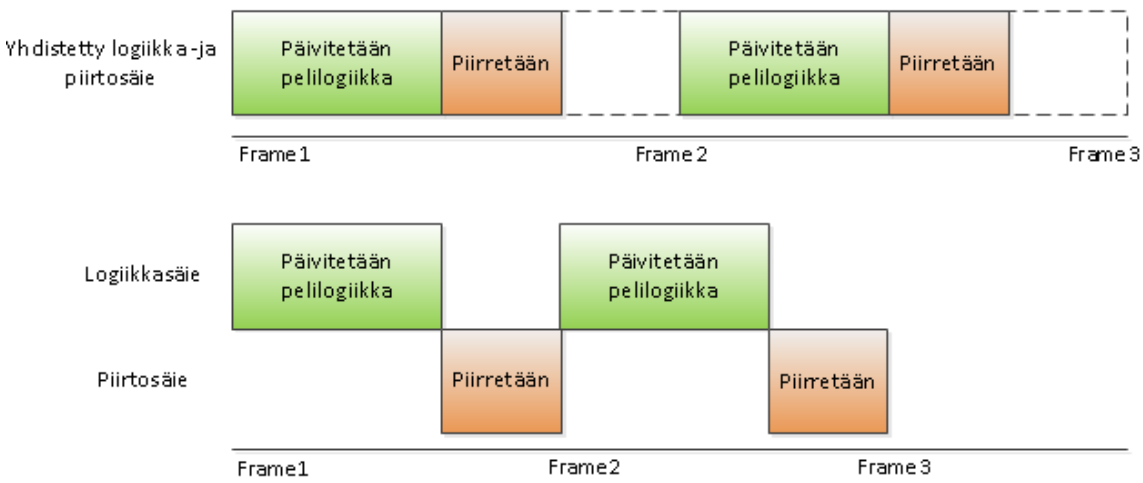
6.3 Säikeistysmalli

Monisäikeistyksen suunnittelu on yksi merkittävimmistä arkkitehtuurillisista valinnoista ohjelmassa kuin ohjelmassa. Järkevästi toteutetulla säikeistyksellä on mahdollista jopa moninkertaistaa sovelluksen suorituskyky. Käytännössä kaikissa pöytäkoneissa on nykyään moniydinprosessori, ja yllättävää kyllä myös mobiililaitteissa tämä alkaa olla enemmän normi kuin poikkeus. Täten mahdollisen monisäikeistyksen tarkastelu on oleellista myös mobiililaitteille sovelluksia suunniteltaessa.

Grafiikkamoottorin perimmäinen tehtävä on varsin yksinkertainen: ottaa talteen piirtokutsuja käyttäjältä ja välittää nämä grafiikkasuorittimelle. Tämä ei yleensä ole kovin aikaavievä operaatio eikä aiheuta pullonkaulaa. Teoriassa OpenGL tukee

piirtokutsujen välittämistä useasta eri säikeestä, mutta ainoastaan yksi säie kerrallaan saa pitää OpenGL-kontekstia hallussaan, joten tekemällä näin ei saavuteta mitään [4].

Varsinainen pullonkaulakohta grafiikkarajapintojen käyttämisessä on vaihe, jossa grafiikkasuoritin alkaa käsitellä sille annettuja piirtokomentoja eli aloittaa uuden ruudun piirtämisen. Piirtämisen aloittamisen ja valmistumisen välillä annetut uudet piirtokomennot jumittavat komentoja antavan säikeen, kunnes grafiikkasuoritin on saanut edellisen ruudun piirrettyä. Ongelma tulee vielä selkeämmin esiin käytettäessä vsync-toimintoa, jolloin ruudunpäivitysnopeus on rajoitettu näyttölaitteen maksimiruudunpäivitysnopeuteen, joka on yleensä 60 ruutua sekunnissa. Tällöin uusi ruutu valmistuu minimissään n. 17 millisekunnin välein, josta siis piirtosäie saattaa olla suuren osan tekemättä mitään. Kuvan 5 ylälaidassa on esitetty kyseinen ongelma tilanteessa, jossa grafiikkamoottoria ajetaan muun ohjelman kanssa samassa säikeessä.



Kuva 5. Yksi -ja kaksisäikeinen toteutus

Edellä kuvattu ongelma voidaan ratkaista jakamalla grafiikkamoottori kahteen säikeeseen: yksi säie syöttää piirtokomentoja puskuriin, josta toinen säie lukee ne ja suorittaa varsinaisen piirron. Näin piirtävä säie jää odottamaan piirron valmistumista, ja piirtokomentoja syöttävä säie pystyy jatkamaan toimintaansa, kuten kuvan 5 alalaidassa ilmenee. Tämä toteutettiin grafiikkamoottoriin *RenderList*-luokan avulla, joka sisältää piirtokomentopuskurin. Käyttäjän antamat piirtokomennot menevät puskuriin, joka annetaan varsinaiselle piirtosäikeelle synkronointipisteessä jokaisen piirretyn ruudun jälkeen.

Eräs toinen käyttökohde säikeistykselle grafiikkamoottoreissa on resurssien lataaminen. Tekstuurien, verteksilistojen ja varjostinohjelmien lataaminen grafiikkasuorittimen muistiin on varsin hidas operaatio, ja mikäli tämä tehdään piirtosäikeessä, voi ruudunpäivityksessä ilmetä havaittavia taukoja. OpenGL tukee resurssien jakamista useiden säikeiden kesken, joten teoriassa on mahdollista siirtää hitaat latausoperaatiot omaan säikeeseensä. Mikäli grafiikkamoottorin olisi tarpeellista tukea ajonaikaista resurssien lataamista, olisi tämä mahdollinen toteutustapa. Tässä grafiikkamoottorissa tällaista toiminnallisuutta ei kuitenkaan tarvita, koska resurssit on tarkoitus ladata ohjelman alussa, joten mahdollisilla hidastumisilla ei ole väliä.

6.4 Piirtokomentojono

Grafiikkamoottorin ytimessä on osa, joka ottaa vastaan piirtokutsut ja välittää nämä grafiikkasuorittimelle. Yksinkertaisimmillaan tämä voi tarkoittaa sitä, että käskettäessä grafiikkamoottoria piirtämään jotain kutsutaan samassa säikeessä suoraan grafiikkarajapinnan piirtokomentoja. Menetelmä on yksinkertainen, mutta sisältää monia haittapuolia. Aiemmin mainittu piirtosäikeen mahdollinen jumiutuminen piirron ajaksi on eräs ongelma, mutta vieläkin isompana ongelmana voidaan pitää sitä, että näin tekemällä peräkkäisiä samanlaisia piirtokomentoja ei pystytä kunnolla optimoimaan. Piirtokutsujen välitön suorittaminen ei tule myöskään kysymykseen sen takia, koska grafiikkamoottori päätettiin jakaa kahteen säikeeseen.

Suoraa piirtämistä parempi vaihtoehto on puskuroida piirtokomennot ennen niiden suorittamista. Puskuroinnilla tarkoitetaan sitä, että piirtokomennot lisätään listaan tai muuhun tietorakenteeseen, josta ne myöhemmin luetaan ja suoritetaan. Tekemällä näin saavutetaan useita hyötyjä:

- Piirtokomennot on helposti lähetettävissä toiseen säikeeseen prosessoitavaksi
- Komentolista voidaan uudelleenjärjestää grafiikkasuorittimen tilavaihtojen minimoimiseksi
- Läpinäkyvät kappaleet saadaan piirtymään oikein järjestämällä komentolista kappaleen etäisyyden perusteella kamerasta

6.4.1 Piirtokomentojen uudelleenjärjestäminen

Piirtokomentojonon uudelleenjärjestäminen turhien tilavaihtojen vähentämiseksi on kätevä tapa hyödyntää listarakennetta. Idea on yksinkertainen: mikäli piirtokomentojonossa on useita samoja piirtoresursseja (tekstuurit, geometria, varjostinohjelma ym.) käyttäviä komentoja, voidaan nämä järjestää esiintymään peräkkäin listassa, jolloin jonoa prosessoitaessa grafiikkasuorittimen tilaa ei tarvitse vaihtaa suoritettaessa näitä identtisiä piirtokomentoja. Näin tilavaihtojen optimointi voidaan piilottaa grafiikkamoottorissa hyvin matalalle tasolle, eikä moottorin käyttäjän tarvitse välittää siitä, missä järjestyksessä piirtokomennot antaa.

Piirtokomentojonon järjestämiseen voidaan käyttää ohjelmointikielen tarjoamia normaaleja lajittelualgoritmeja, esim. C++:aa käytettäessä voidaan käyttää standardikirjaston *std::sort*-funktiota. Lajittelualgoritmillle on annettava vertailufunktio, jonka avulla se kykenee järjestämään piirtokomennot. Vertailufunktio määrittää mihin järjestykseen ja millä perusteella piirtokomentojen halutaan järjestyvän.

Koska piirtokomentojonon lajittelu tehdään jokaisella piirretyllä ruudulla uudelleen, ja monimutkaisissa 3D-näkymissä piirtokomentoja saattaa olla useita satoja, halutaan lajittelun olevan mahdollisimman nopea operaatio. Ilmenee, että piirtokomentojen lajittelu on mahdollista optimoida kokonaislukulajitteluksi määrittämällä piirtokomennolle avain, joka kuvaa piirtokomennon käyttämiä resursseja. Avain on kokonaisluku, josta varataan bittejä ilmaisemaan komennon käyttämiä resursseja. Lajiteltaessa avaimet suuruusjärjestykseen samoja resursseja käyttävät avaimet päätyvät automaattisesti peräkkäin. Itse piirtokomentoja ei siis tarvitse lajitella lainkaan, vaan piirtosäikeen alkaessa käsittelemään piirtokomentojonoa komentojen järjestys luetaan lajitellusta avainlistasta. Eri resurssityyppien priorisointia on helppoa muuttaa varaamalla resurssia merkitsevät bitit joko korkeammalta tai matalammalta kokonaislukuavainta. Kyseinen tekniikka on käytössä myös joissain kaupallisissa pelimoottoreissa [5]. Listauksessa 2 on esitettynä tähän grafiikkamoottoriin valittu piirtokomentojen lajitteluavain.

```
union RenderCallKey {
            struct {
                        unsigned int unused : 5;
                        unsigned int texture1 : 7;
                        unsigned int texture0 : 7;
                        unsigned int program : 5;
                        unsigned int group : 4;
                        unsigned int blending : 2;
                        unsigned int command : 1;
                        unsigned int layer : 1;
            };
            struct {
                        unsigned int depth : 24;
                        unsigned int group : 4;
                        unsigned int blending : 2;
                        unsigned int command : 1;
                        unsigned int layer : 1;
            } transparent;
            unsigned int dword;
};
```

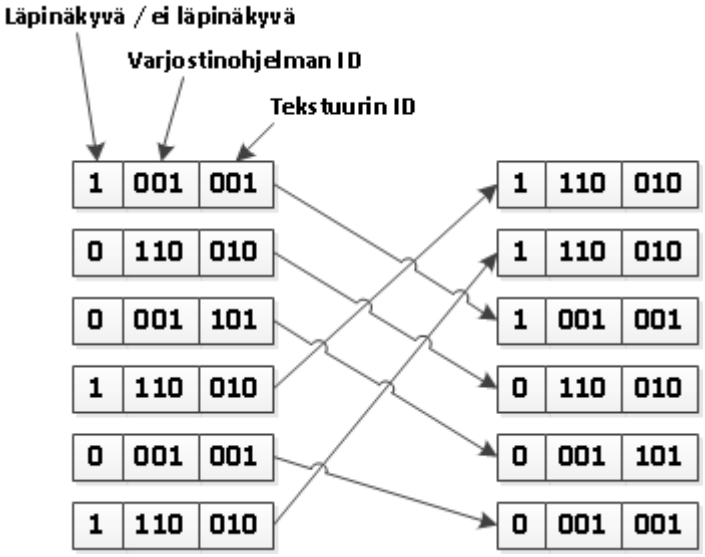
Koodiesimerkki 2. Piirtokomentoavain

6.4.2 Läpinäkyvä geometria

Osittain läpinäkyvän 3D-geometrian saaminen piirtymään visuaalisesti oikein on yllättävän hankalaa. Ongelma kumpuaa tavasta, jolla syvyyspuskuri toimii. Piirrettäessä kappaletta sen pikselien etäisyys kamerasta tallennetaan syvyyspuskuriin. Seuraavaa kappaletta piirrettäessä tämän pikselien etäisyyttä verrataan syvyyspuskurissa jo oleviin arvoihin. Ainoastaan lähempänä kameraa olevat uuden kappaleen pikselit päätyvät näytölle. Tämä mahdollistaa läpinäkymättömien kappaleiden piirtämisen missä järjestyksessä tahansa, ilman että kappaleiden piirtokutsuja olisi järjestettävä etäisyyden mukaan. Valitettavasti idea ei ollenkaan toimi piirrettäessä läpinäkyviä kappaleita. Mikäli lähelle kameraa piirretään läpinäkyvä kappale, ja tämän jälkeen yritetään sen taakse piirtää toinen läpinäkyvä kappale, ei jälkimmäinen näy lopullisessa kuvassa ollenkaan edellä mainitusta syvyyspuskuritarkistuksesta johtuen. Jotta läpinäkyvät kappaleet saadaan piirtymään oikein, on syvyyspuskuritarkistus otettava pois päältä, ja piirrettävä kappaleet etäisyyden perusteella kamerasta aloittaen kauimmaisesta.

Läpinäkyvien kappaleiden piirtokomentojen lajittelu etäisyyden perusteella voidaan kätevästi yhdistää osaksi piirtokomentojonon uudelleenjärjestämistä käyttämällä osa lajitteluavaimen biteistä kappaleen etäisyyden ilmaisemiseen. Piirtokomentojonossa voi olla samaan aikaan sekä tavanomaisia piirtoresurssien perusteella järjestettyjä

piirtokomentoja, että etäisyyden perusteella järjestettyjä piirtokomentoja. Tämän mahdollistaa lajitteluavaimessa oleva piirtokomennon tyyppiä ilmaiseva *blending*-kenttä, jonka arvo on eri resurssien perusteella järjestetyille komennoille ja etäisyyden perusteella järjestetyille. Koska tämä kenttä on korkeammalla avaimessa kuin piirtoresurssit tai etäisyys, päätyvät erityyppiset piirtokomennot omiin lohkoihinsa jonossa, ja näin piirtojärjestys pysyy oikeana. Kuva 6 havainnollistaa erityyppisten piirtokomentojen avaimien järjestämistä.



Kuva 6. Piirtokomentojonon uudelleenjärjestäminen

6.4.3 Muut komennot

Piirtokomentojonoon on mahdollista lisätä myös muunlaisia komentoja piirtokäskyjen lisäksi. Tällaisia voisivat olla esimerkiksi grafiikkasuorittimen tilan muuttaminen jotain erityisefektiä varten tai varjostinohjelman parametrien välitys piirtosäikeeseen. Lajitteluavaimesta varataan yksi bitti ilmaisemaan sitä, onko komento piirtokäsky vai muu komento (*command*-kenttä listauksessa 2). Loput bitit voidaan käyttää komennon tyypin ja mahdollisten parametrien ilmaisemiseen.

6.5 Varjostinohjelmien hallinta

Varjostinohjelmat ovat keskeisessä asemassa modernissa 3D-grafiikkamoottorissa. Kuten luvussa 2 mainitaan, nykyajan grafiikkasuorittimet piirtävät kaiken varjostinohjelmia käyttäen, olipa kyseessä sitten yksinkertainen 3D-malli tai monimutkainen, useita piirtokertoja ja eri varjostinohjelmia yhdistelevä graafinen erikoisefekti. Jotta varjostinohjelman toimintaa saadaan konfiguroitua, eikä kaikkia ohjelman tietoja tarvitse kovakoodata, voidaan varjostinohjelmalle välittää parametreja ohjelmallisesti. Monimutkaisissa 3D-peleissä erilaisia varjostinohjelmia voi olla satoja, jolloin hallittavan datan määrä kasvaa valtavaksi. Tästä syystä varjostinohjelmien ja niiden parametrien hallintaan tarvitaan järjestelmä, joka huolehtii parametrien asettamisesta oikeaan aikaan automaattisesti.

Jotta varjostinohjelmien tarvitsemista tiedoista saadaan parempi käsitys, tarkastellaan esimerkkinä yksinkertaista verteksivarjostinohjelmaa (listaus 1). Kyseinen varjostinohjelma suorittaa koordinaattimuunnoksen 3D-mallin vertekseille siirtäen ne mallin paikallisesta koordinaatistosta 3D-maailman koordinaatistoon. Varjostinohjelma tarvitsee kaksi parametria: *uViewProjMatrix* ilmaisee kameran sijainnin, katselukulman ja muut asetukset; *uWorldMatrix* ilmaisee 3D-mallin sijainnin ja asennon 3D-maailmassa. Ensiksi mainittu parametri on riippumaton piirrettävästä objektista. Sen arvo pysyy samana, piirrettiin sitten yksi tai sata 3D-mallia. Jälkimmäinen parametri sen sijaan on uniikki jokaiselle piirrettävälle objektille. Havaitaan, että varjostinohjelmien parametrit voidaan jaotella kahteen eri tyyppiin: vakioparametreihin ja muuttuviin parametreihin. Vakioparametrien arvot asetetaan vain kerran jokaisella piirrettävällä ruudullisella. Näin säästetään grafiikkasuorittimelle lähetettävän datan määrässä. Muuttuvat parametrit sen sijaan on lähetettävä uudelleen jokaisen piirtokomennon yhteydessä.

Vakioparametrien asettaminen toteutettiin grafiikkamoottoriin seuraavalla tavalla: vakioparametreille on olemassa yksi yhteinen tietorakenne (listaus 3), joka sisältää kaikkien varjostinohjelmien tarvitsemat vakioparametrit. Jokaisella piirretyllä ruudulla tietorakenteeseen asetetaan halutut arvot, ja lisätään se piirtokomentojonoon. Nyt parametrit ovat automaattisesti kaikkien varjostinohjelmien käytettävissä.

```
struct OnetimeShaderParams {
            Matrix4f viewProjMatrix;
            Vector3f fogColor;
            Vector2f fogRange;
            Vector3f lightDir;
            Vector3f lightColor;
            Vector3f ambientColor;
            float heightOfNearPlane;
};
```

Koodiesimerkki 3. Tietorakenne varjostinohjelmien vakioparametreille

Muuttuvat parametrit välitetään piirtokomentojen yhteydessä. Listauksessa 4 on esitetty uuden piirtokomennon lisääminen komentojonoon, jonka jälkeen muuttuvien parametrien arvot asetetaan.

```
ModelShaderParams* modelParams =
renderList.addCall<ModelShaderParams>(mdl->getObject(0).renderCall,
                                      mdl->getObject(0).renderCallKey);
modelParams->worldMatrix.identity();
modelParams->worldMatrix.setRotation(_world._playerRot);
modelParams->worldMatrix.setTranslation(_world._playerPos);
```

Koodiesimerkki 4. Muuttuvien varjostinohjelmaparametrien asettaminen

Parametrien automatisoinnista johtuen uusien varjostinohjelmien lisääminen grafiikkamoottoriin on vaivatonta. Varjostinohjelman latausalgoritmi parsii varjostinohjelman etsien tunnettuja parametrien nimiä ja ottaa ne käyttöön automaattisesti. Lisättäessä uusia varjostinohjelmia grafiikkamoottorin koodiin ei siis tarvitse tehdä mitään muutoksia, paitsi ainoastaan siinä tilanteessa, mikäli varjostinohjelman käyttöön halutaan tuoda täysin uusi parametri. Tässä tapauksessa se on lisättävä joko vakioparametrien tietorakenteeseen tai rekisteröitävä muuttuvaksi parametriksi ja asetettava piirtokomennon yhteydessä.

6.6 Oma 3D-malliformaatti

3D-grafiikan tuottamiseen on useita tapoja. Tarpeeksi yksinkertainen 3D-geometria voidaan generoida ohjelmallisesti suoritusaikana. Yleensä kuitenkin 3D-grafiikan luomiseen käytetään 3D-mallinnus- ja animointiohjelmistoja, jotka tuottavat 3D-mallin geometrian ja muut ominaisuudet sisältävän tiedoston. Grafiikkamoottori lataa

tiedoston ja muuntaa sen sisältämän datan grafiikkarajapinnan ymmärtämään muotoon. On olemassa monia valmiita 3D-malliformaatteja, joita 3D-mallinnusohjelmistot yleisesti tukevat. Eräitä esimerkkejä ovat Autodesk 3ds Max -mallinnusohjelman käyttämä 3DS-formaatti sekä laajalti käytetty Wavefront OBJ-formaatti. Yksi mielenkiintoisimmista on OpenGL:n kehittäjän Khronos Group:n luoma COLLADA-formaatti. Se on avoin XML-pohjainen formaatti, ja tukee valtavaa määrää erilaisia ominaisuuksia. Se on myös hyvin laajalti tuettuna mallinnusohjelmissa sekä monissa valmiissa peli- ja grafiikkamoottoreissa [6].

Valmiiden 3D-malliformaattien käyttöön liittyy joitain ongelmia. Mikäli formaatti on binäärimuotoinen, voi sen lataaminen olla hankalaa. Toisaalta mikäli formaatti on teksti- tai XML-pohjainen, voi 3D-mallien tiedostokoko kasvaa turhan suureksi. On myös mahdollista, että formaatti ei tue jotakin kaivattua ominaisuutta, tai toisaalta tukee ominaisuuksia, joita käyttötapauksessa ei kaivata, kasvattaen näin turhaan tiedostokokoa. Näistä syistä monissa grafiikka- ja pelimoottoreissa on käytössä oma 3D-malliformaatti. Tällä tavoin kehittäjät voivat määrittää täsmälleen, mitä ominaisuuksia haluavat tukea ja voivat optimoida formaatin sisäisen rakenteen omaan moottoriinsa sopivaksi.

Jotta omaa 3D-malliformaattia varten ei tarvitse kehittää myös omaa mallinnusohjelmaa, tarvitaan muunnostyökalu, joka muuntaa jonkin tunnetun 3D-malliformaatin omaan formaattiin. Toinen vaihtoehto on kirjoittaa mallinnusohjelmalle export-skripti, jolla mallinnusohjelma saadaan suoraan tuottamaan malleja omassa formaatissa. Kaikki suosituimmat 3D-mallinnusohjelmat tukevat jonkinasteista skriptaamista, jolla tämän pitäisi onnistua. Esimerkiksi avoimen lähdekoodin Blender-mallinnusohjelma on hyvä lähtökohta. Blender tukee Python-ohjelmointikielisiä skriptejä ja plugineja, ja sisältää hyvin dokumentoidun rajapinnan, joten kaikenlaisten lisäosien tekeminen on vaivatonta.

Grafiikkamoottoria suunniteltaessa päädyttiin oman 3D-malliformaatin kehittämiseen, lähinnä edellä mainituista syistä. Koska moottori on mobiililaitteita varten kehitetty, halutaan 3D-mallien latautuvan mahdollisimman nopeasti ja vähän resursseja käyttäen. Formaatin sisäinen rakenne on suunniteltu siten, että 3D-geometrian sisältävät tietorakenteet voidaan suoraan ladata OpenGL:lle ilman esikäsittelyä. Formaatin tukemien ominaisuuksien määrä rajattiin hyvin pieneksi, esimerkiksi minkäänlaista

tukea animaatioille ei ole. Formaattia varten kirjoitettiin Blender-mallinnusohjelmalle export-skripti. Formaatin pseudokielinen kuvaus löytyy liitteestä 1.

7 Tekniikkademo

Grafiikkamoottorin ohessa on kehitetty tekniikkademoa, joka käyttää moottoria grafiikkansa piirtämiseen. Alun perin tarkoituksena oli kehittää toimiva mobiilipeli grafiikkamoottoria käyttäen, mutta ajanpuutteen ja muiden syiden vuoksi idea kariutui. Pelin jäänteistä alettiin koota tekniikkademoa, jolla moottorin ominaisuuksia voidaan testata ja esitellä. Tekniikkademo käyttää suurta osaa moottorin tarjoamista ominaisuuksista, ja on siten havainnollistava osoitus siitä, mihin moottori kykenee. Seuraavaksi tekniikkademoa esitellään tarkemmin ja näytetään koodiesimerkein, miten grafiikkamoottoria on siinä käytetty.

Tekniikkademo muistuttaa lentosimulaattoria, koska kehitettävästä pelistä oli tarkoitus tulla jotain tämän tapaista, kuten kuvasta 7 havaitaan. Tekniikkademossa on yksinkertainen, heightmap-tekniikalla toteutettu maastogeometria. Koska tarkoituksena oli tehdä mobiililaitteille lentosimulaattori, on maaston resoluutio erittäin matala: yksi verteksi 50 metrin pelimaastoa kohden. Näin matala resoluutio aiheuttaa ongelmia maaston valaistuksessa (kuvassa näkyvät tummat kohdat maastossa), koska verteksejä, ja siten myös pinnan normaaleja on niin harvassa ja näytönohjain joutuu interpoloimaan ne pikselikohtaisesti. Maaston maksimikokoa ei ole rajoitettu, vaan koon rajoittaa käytettävissä oleva muisti. Tekniikkademon käyttämä maasto on kooltaan 256 x 256 verteksiä (eli pelimaailman mitoissa 12,8 km x 12,8 km), mutta vie vain reilun megatavun muistia. Maaston resoluutiota voitaisiinkin luultavasti nostaa suorituskyvyn juuri kärsimättä.

Kuva 7. Tekniikkademo

Maastogeometria ladataan näytönohjaimelle käyttäen matalan tason VertexBufferManager- ja IndexBufferManager-luokkia. Maaston käyttämä tekstuuri ladataan TextureManager-luokalla. Listauksessa 5 on esitetty nämä toimenpiteet. Maaston piirtäminen hoidetaan samalla tavalla kuin kaikki piirtäminen, eli kutsumalla RenderList-luokan *addRenderCall*-metodia, joka lisää piirtokutsun piirtokomentojonoon.

```
gfx::VertexTerrainHigh* vertices =
                        new gfx::VertexTerrainHigh[numVertsInChunk];
// Generoidaan verteksidata
// ...
_vbo = _services.getVertexBufferManager()
            .createBuffer(gfx::VT_TERRAINHIGH, gfx::BU_STATIC,
                                    numVertsInChunk, vertices);

unsigned short* indices = new unsigned short[numIndices];
// Generoidaan indeksit
// ...
_ibo = _services.getIndexBufferManager()
            .createBuffer(gfx::BU_STATIC, numIndices, indices);
```

Koodiesimerkki 5. Grafiikkamoottorin matalan tason palveluiden käyttäminen

Kuvassa 7 näkyvä lentokone käyttää luvussa 6.6 kuvattua omaa 3D-malliformaattia. 3D-mallien lataamista varten on olemassa ModelManager-luokka, joka hoitaa sisäisesti raakojen grafiikkaresurssien (verteksipuskurit ym.) luomisen. ModelManager-luokalta

saadaan Model-instanssi, joka sisältää listan 3D-objekteja. Nämä objektit sisältävät piirtokutsun, ja ne voidaan piirtää RenderList-luokan avulla. Listauksessa 6 on esitetty 3D-mallin lataaminen ja piirtäminen.

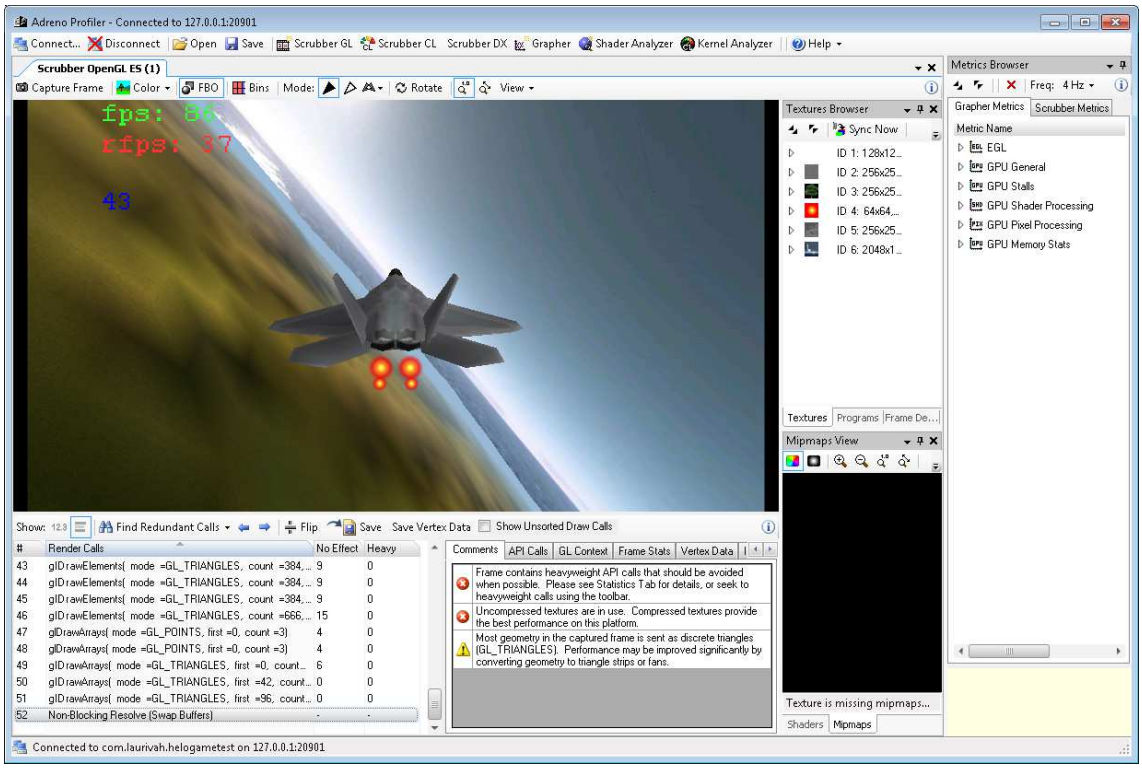
```
// Ladataan 3D-malli
Model* mdl = _modelMgr.load("f22.3dm");

// Piirretään ja asetetaan muuttuvat varjostinohjelmaparametrit
ModelShaderParams* modelParams = renderList.addCall<ModelShaderParams>(
                                    mdl->getObject(0).renderCall,
                                    mdl->getObject(0).renderCallKey);
modelParams->worldMatrix.identity();
modelParams->worldMatrix.setRotation(_world._playerRot);
modelParams->worldMatrix.setTranslation(_world._playerPos);
```

Koodiesimerkki 6. 3D-mallin lataaminen ja piirtäminen

8 Suorituskykymittaukset

Grafiikkamoottorilla suoritettiin erinäisiä suorituskykymittauksia, joiden tavoitteena oli arvioida moottorin rakenteen onnistumista suorituskykyä ajatellen sekä sen skaalautuvuutta erilaisia 3D-maailmoja esitettäessä. Vertailukohtana käytetään testilaitteen grafiikkapiirin keskimääräisiä suoritusarvoja. Ilmoitetut arvot ovat toki teoreettisia ja saavutettavissa vain täysin synteettisiä testejä ajettaessa, mutta ne antavat silti jonkinlaisen käsityksen grafiikkamoottorin suorituskyvystä.

Mittaukset suoritettiin kirjaamalla testilaitteen ruudunpäivitysnopeutta erilaisia grafiikkakuormia ajettaessa. Yksinkertaisen FPS-arvon kirjaamisen lisäksi mittauksissa käytettiin Qualcomm:n kehittämää Adreno Profiler -grafiikkaprofilointityökalua (kuva 8). Työkalua on mahdollista käyttää kahdella tavalla. Grapher-tilassa ohjelmalla pystyy seuraamaan grafiikkapiirin reaaliaikaisia suoritusarvoja, kuten ruudunpäivitysnopeutta, varjostinyksiköiden laskentakuormaa sekä muistinkäyttöä. Scrubber-tilassa ohjelmalla ”kaapataan” yksittäinen piirretty ruutu, jonka jälkeen kyseisellä ruudulla annettuja piirtokomentoja voidaan vapaasti tarkastella ja muokata sekä tutkia näiden vaikutusta suorituskykyyn. Tässä tilassa ohjelmalla pystytään myös katselemaan ja muokkaamaan käytössä olevia varjostinohjelmia.



Kuva 8. Adreno Profiler -grafiikkaprofilointityökalu

Mittauksissa käytettiin testilaitteena luvussa 5.3 esiteltyä Android-älypuhelinta. Laitteessa on Qualcommin valmistama Adreno 205 -grafiikkapiiri. Piiri toimii 400 MHz:n nopeudella, ja on huomattavasti aiempaa Adreno 200 -piiriä tehokkaampi etenkin varjostinohjelmien laskentatehossa [8]. Piirin maksimaalinen teoreettinen verteksiprosessointinopeus on 57 miljoonaa kolmiota sekunnissa, ja pikseliprosessointinopeus on 539 miljoonaa pikseliä sekunnissa [9]. Mikäli tähdätään ruudunpäivitysnopeuteen 30 ruutua sekunnissa, tekee tämä n. 2 milj. kolmiota ja 18 milj. piirrettyä pikseliä ruudullisella.

Ensimmäisessä mittauksessa mitattiin piirrettyjen kolmioiden määrän vaikutusta ruudunpäivitysnopeuteen. Näytölle piirrettiin 3D-malli, joka koostui n. 1000 kolmiosta ja mitattiin Adreno Profilerilla ruudunpäivitysnopeus sekä tarkistettiin piirrettyjen kolmioiden ja pikselien määrä. Mittaus toistettiin viidellä eri määrällä 3D-malleja lähtien sadasta ja päätyen kahteentuhanteen. Kyseiset testiajot suoritettiin ensin grafiikkamoottoria käyttäen, jonka jälkeen täysin vastaavanlaiset testit ajettiin pelkkää OpenGL:ää käyttäen. Näin voitiin arvioida grafiikkamoottorin abstraktioiden aiheuttamaa ”overheadia” suhteessa suoraan grafiikkarajapinnan käyttämiseen. Liitteessä 2 on esitetty saadut mittaustulokset.

Kuten mittaustuloksista havaitaan, jäätiin sekä grafiikkamoottoria että pelkkää OpenGL:ää käytettessä hyvin kauas grafiikkasuorittimen teoreettisesta maksimisuorituskyvystä. Oli odotettavaa, että maksimisuorituskyvystä jäädään jonkin verran jälkeen, mutta havaittu yli kymmenkertainen ero on kuitenkin hämmentävän suuri. Syitä tähän on voi olla monia. Ilmoitetut arvot ovat todennäköisesti saavutettavissa vain hyvin tarkoin rajatuissa testiolosuhteissa, mahdollisesti ajettaessa testejä suoraan grafiikkasuorittimen kanssa kommunikoiden, ilman käyttöjärjestelmän ja grafiikkarajapintojen aiheuttamaa kuormaa. Joka tapauksessa tuloksista voidaan kuitenkin havaita, että suorituskykyero grafiikkamoottorilla ja pelkällä OpenGL:llä ajettujen testien välillä on mitätön. Merkittävää eroa ei havaita edes suurilla piirtokomentomäärillä. Tästä voidaan vetää johtopäätös, että grafiikkamoottorin abstraktiot ovat varsin onnistuneita.

Toisessa mittauksessa mitattiin piirtokomentojonon järjestämisen vaikutusta ruudunpäivitysnopeuteen. Piirtokomentojonon järjestämisen ideana on vähentää OpenGL:lle välitettäviä turhia tilanvaihtoja ja näin parantaa ruudunpäivitysnopeutta. Mittaus toteutettiin piirtämällä testinäkymä, jossa on kymmentä eri tekstuuria ja viittä eri varjostinohjelmaa käyttäviä 3D-malleja, eli yhteensä eri kombinaatioita on 100. 3D-mallit piirrettiin satunnaisessa järjestyksessä, jotta tilanvaihtojen määrä saatiin mahdollisimman suureksi. Testinäkymää ajettiin ensin ilman piirtokomentojonon järjestämistä ja sen jälkeen järjestäminen päällä. Samalla mitattiin ruudun piirtämiseen kuluvan ajan (eng. *frametime*) keskiarvoa. Saadut mittaustulokset on esitetty liitteessä 3 sekä tuloksista piirretyt kuvaajat liitteessä 4.

Kuten mittaustuloksia tarkastelemalla voidaan havaita, on piirtokomentojonon järjestämisellä todellinen, joskin varsin pieni vaikutus ruudunpäivitysnopeuteen. Pienillä piirtokomentomäärillä lajittelun vaikutus on mitätön ja pienentää ruudun piirtoaikaa alle millisekunnilla. Piirtokomentomäärän noustessa useisiin satoihin alkaa lajittelun merkitys kasvaa. 500 piirtokomennolla lajittelu parantaa piirtoaikaa jo n. 10 millisekunnilla. Näin suurilla piirtokomentomäärillä alkaa kuitenkin grafiikkasuorittimen suorituskyky ottamaan vastaan, ja ruudunpäivitysnopeus alenee kelvottomiin lukemiin reaaliaikaista toimintaa ajatellen. Näiden tulosten valossa voidaankin koko piirtokomentojonon lajittelun merkitys kyseenalaistaa. On kuitenkin pidettävä mielessä, että piirtokomentojonon lajittelulla on grafiikkamoottorissa piirron optimoinnin ohella myös muita tehtäviä (ks. luku 6.4).

9 Yhteenveto

Projektin tavoitteena oli toteuttaa yksinkertainen mobiililaitteilla toimiva 3D-grafiikkamoottori, ja samalla tutustua 3D-grafiikkaan ja OpenGL-rajapintaan. Grafiikkamoottoria käyttäen toteutettiin myös hyvin suppea tekniikkademo, ja lopuksi mitattiin grafiikkamoottorin suorituskykyä.

Projektin päätavoitteiden voidaan katsoa saavutetuiksi. Toteutetulla grafiikkamoottorilla pystyy lataamaan ja piirtämään teksturoituja 3D-malleja erilaisia varjostinohjelmia käyttäen. Grafiikkamoottorin rakenteesta tuli kohtuullisen selkeä, ja näin ominaisuuksia on myöhemmin mahdollista lisätä pienellä vaivalla. Etenkin varjostinohjelmien hallintajärjestelmästä tuli varsin onnistunut, ja sen avulla grafiikkamoottoriin on vaivatonta lisätä uusia varjostinohjelmia. Tästä huomattiin olevan käytännön hyötyä jo kehitysvaiheessa suorituskykymittauksien vaatiessa useiden eri varjostinohjelmien käyttämistä.

Alun perin grafiikkamoottoria käyttäen oli tarkoitus toteuttaa toimiva mobiilipeli, mutta ideasta jouduttiin luopumaan ajanpuutteen vuoksi. Tämän seurauksena grafiikkamoottorin käyttötarkoitus ja tuetut ominaisuudet jäivät varsin heikosti määritellyiksi. Onkin kyseenalaista, pystyisikö grafiikkamoottorin tämän hetkisellä versiolla toteuttamaan kokonaisen pelin, johtuen sen hyvin rajallisista ominaisuuksista.

Grafiikkamoottoria toteutettaessa kävi selväksi, kuinka työlästä näin matalan tason ohjelman toteuttaminen on. Grafiikkamoottori joutuu käsittelemään suurta määrää erilaisia grafiikkaresursseja, ja näiden hallinnan suunnittelu vaatii syvällistä ymmärrystä grafiikkarajapinnoista. Ei olekaan ihme, että suurin osa pelintekijöistä päätyy käyttämään valmiita peli- tai grafiikkamoottoreita sen sijaan, että kirjoittaisi kaiken alusta asti itse. Rehellisesti voidaan todeta, että esimerkiksi Unity-pelimoottoria käyttäen olisi tässä työssä esitellyn tekniikkademon kaltaisen sovelluksen toteuttanut parissa tunnissa.

Joka tapauksessa grafiikkamoottorin toteuttaminen oli erittäin opettavaa ja mielenkiintoista. Vaikka toteutettu versio on ominaisuuksiltaan yksinkertainen, tarjoaa se hyvän pohjan jatkokehitykselle.

Lähteet

1 Dashboards, Android Developers. Verkkodokumentti. <http://developer.android.com/about/dashboards/index.html> Luettu 1.1.2014.

2 OpenGL ES Graphics, iOS Developer Library. Verkkodokumentti. <https://developer.apple.com/library/ios/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/OpenGLESPlatforms/OpenGLESPlatforms.html> Luettu 1.1.2014.

3 OpenGL ES, Wikipedia. Verkkodokumentti. <http://en.wikipedia.org/wiki/OpenGL_ES> Luettu 25.1.2014.

4 OpenGL and multithreading. Verkkodokumentti. <http://www.opengl.org/wiki/OpenGL_and_multithreading> Luettu 25.1.2014.

5 Order your graphics draw calls around! realtimecollisiondetection.net - the blog. Verkkodokumentti. <http://realtimecollisiondetection.net/blog/?p=86> Luettu 8.2.2014.

6 Products directory, Collada. Verkkodokumentti. <https://collada.org/mediawiki/index.php/Portal:Products_directory> Luettu 15.2.2014.

7 Early Z Rejection, Intel Developer Zone. Verkkodokumentti. <https://software.intel.com/en-us/vcsource/samples/early-z-rejection> Luettu 29.3.2014.

8 Adreno Graphics Processing Units, Qualcomm Developer Network. Verkkodokumentti. <https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/adreno-gpu> Luettu 29.3.2014.

9 Adreno, Wikipedia. Verkkodokumentti. <http://en.wikipedia.org/wiki/Adreno> Luettu 29.3.2014.

10 Unity rendering, Unity. Verkkodokumentti. <https://unity3d.com/unity/quality/rendering> Luettu 5.4.2014.

11 Features, Ogre. Verkkodokumentti. <http://www.ogre3d.org/about/features> Luettu 5.4.2014.

12 Preface: What is OpenGL? OpenGLBook.com. Verkkodokumentti. <http://openglbook.com/the-book/preface-what-is-opengl> Luettu 25.1.2014.

13 OpenGL ES 1_x. Kuva. <http://www.khronos.org/assets/uploads/apis/opengles_1x_pipeline.gif> Katsottu 25.1.2014.

14 OpenGL ES 2. Kuva. <http://www.khronos.org/assets/uploads/apis/opengles_20_pipeline2.gif> Katsottu 25.1.2014.

15 Instancing, OpenGL.org. Verkkodokumentti. <https://www.opengl.org/wiki/Vertex_Rendering#Instancing> Luettu 29.3.2014.

16 OpenGL Overview. Verkkodokumentti. <http://www.opengl.org/about/> Luettu 25.1.2014.

17 OpenGL Shading Language, Wikipedia. Verkkodokumentti. <http://en.wikipedia.org/wiki/OpenGL_Shading_Language> Luettu 25.1.2014.

18 100 Most Popular Game Engines, Indie DB. Verkkodokumentti. <http://www.indiedb.com/engines/top> Luettu 5.4.2014.

19 About, Ogre. Verkkodokumentti. <http://www.ogre3d.org/about> Luettu 5.4.2014.

20 Ogre Android. Verkkodokumentti. <http://www.ogre3d.org/tikiwiki/Ogre+Android> Luettu 5.4.2014.

21 Irrlicht Engine. Verkkodokumentti. <http://irrlicht.sourceforge.net/> Luettu 5.4.2014.

22 Features, Irrlicht Engine. Verkkodokumentti. <http://irrlicht.sourceforge.net/features/> Luettu 5.4.2014.

23 Android NDK, Android Developers. Verkkodokumentti. <https://developer.android.com/tools/sdk/ndk/index.html> Luettu 29.3.2014.

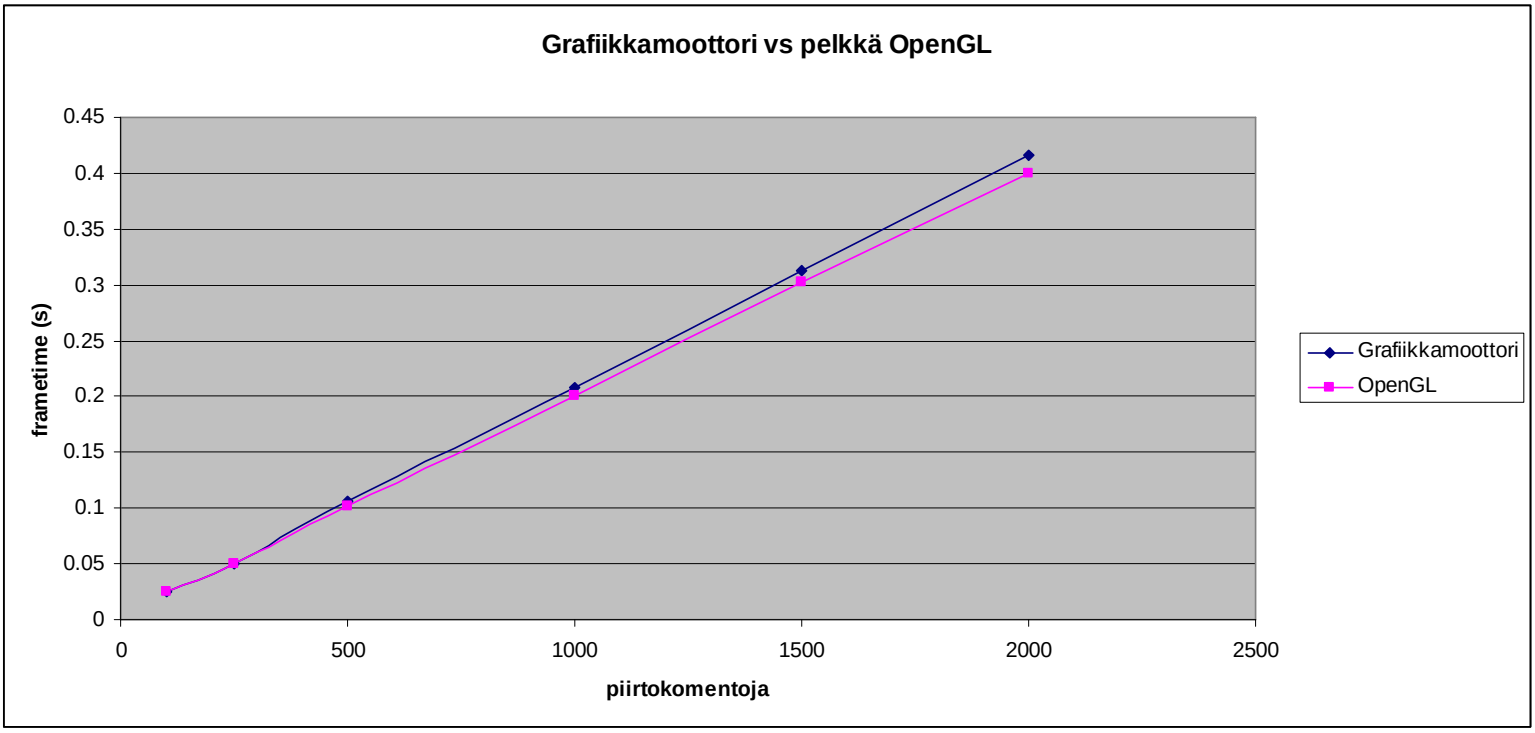
3D-malliformaatin kuvaus

Seuraava pseudokielinen kuvaus esittää grafiikkamoottorissa käytettävän 3D-malliformaatin rakenteen.

```
struct Model {
            char signature[4]; // Signature of the file, string "MODL"
            unsigned int version; // Version of the model format
            unsigned int numTextures; // Number of textures used in the model
            BinString textureList[numTextures]; // List of texture names
            unsigned int numObjects; // Number of objects in the model
            Object objectList[numObjects]; // List of objects
};

struct BinString {
            unsigned short numReserved;
            unsigned short numCharacters;
            char characters[numReserved];
};

struct Object {
            unsigned int id; // ID of the object
            struct ObjectFlags {
                        int zTestMode : 3; // Depth test mode
                        int zWrite : 1; // Enable depth buffer write
                        int culling : 1; // Enable culling
                        int group : 4; // Group ID
                        int blendMode : 2; // Blending mode
                        int shader : 5; // Shader ID
            } flags;

            unsigned short textureId; // ID of the texture
            unsigned short numVertices; // Number of vertices
            unsigned short numIndices; // Number of indices
            float offset[3]; // Position offset of the object

            struct Vertex {
                        float x, y, z;
                        char nx, ny, nz, padding;
                        unsigned short u, v;
            } vertices[numVertices];

            unsigned short indices[numIndices];

            // Present only if numIndices % 2 == 1,
            // to align the data to a 32-bit boundary
            unsigned short padding;


};
```

Suorituskykymittaus 1, taulukoidut tulokset ja kuvaaja

Piirtokomentojen määrä	Piirrettyjä kolmoita (Mtris/frame)	Piirrettyjä pikseleitä (Mpixels/frame)	Grafiikkamoottori, frametime (s)	Pelkkä OpenGL, frametime (s)
100	0,097	0,487	0,025188917	0,024390244
250	0,242	0,660	0,050505051	0,05
500	0,484	0,920	0,106382979	0,102040816
1000	0,968	1,501	0,208333333	0,2
1500	1,452	2,004	0,3125	0,303030303
2000	1,936	2,552	0,416666667	0,4



Suorituskykymittaus 2, taulukoidut tulokset

Mittaustulokset käytettäessä 3D-mallia, jossa n. 200 verteksiä

Piirtokomentojen määrä	**Lajittelu pois käytöstä, frametime (s)**	**Lajittelu käytössä, frametime (s)**	**Erotus (s)**
100	0,023255814	0,023255814	0,0
250	0,041666667	0,037735849	0,003930818
500	0,078125	0,06993007	0,00819493
1000	0,181818182	0,153846154	0,027972028
1500	0,285714286	0,238095238	0,047619048
2000	0,384615385	0,303030303	0,081585082

Mittaustulokset käytettäessä 3D-mallia, jossa n. 500 verteksiä

Piirtokomentojen määrä	**Lajittelu pois käytöstä, frametime (s)**	**Lajittelu käytössä, frametime (s)**	**Erotus (s)**
100	0,032258065	0,031446541	0,000811524
250	0,066666667	0,063291139	0,003375527
500	0,142857143	0,133333333	0,00952381
1000	0,285714286	0,25	0,035714286
1500	0,454545455	0,4	0,054545455
2000	0,625	0,5	0,125

Suorituskykymittaus 2, kuvaajat